Fast CRCs

Gam D. Nguyen

Abstract—CRCs have desirable properties for effective error detection. But their software implementation, which relies on many steps of the polynomial division, is typically slower than other codes such as weaker checksums. A relevant question is whether there are some particular CRCs that have fast implementation. In this paper, we introduce such fast CRCs as well as an effective technique to implement them. For these fast CRCs, even without using table lookup, it is possible either to eliminate or to greatly reduce many steps of the polynomial division during their computation.

Index Terms—Fast CRC, low-complexity CRC, checksum, error-detection code, Hamming code, period of polynomial, fast software implementation.

1 Introduction

This paper considers cyclical redundancy checks (CRCs), which are effective for detecting errors in communication and computer systems. An h-bit CRC is typically generated by a binary polynomial of the form

$$M(X) = (X+1)M_1(X), \quad (1)$$

where $M_1(X)$ is a primitive polynomial of degree $h-1$. Existing CRCs include the CRC-16 generated by $X^{16}+X^{15}+X^2+1=(X+1)(X^{15}+X+1)$, and the CRC-CCITT generated by $X^{16}+X^{12}+X^5+1=(X+1)(X^{15}+X^{14}+X^{13}+X^{12}+X^4+X^3+X^2+X+1)$.

The CRC generated by (1) has the following desirable properties: 1) its maximum length is $2^{h-1}-1$ bits, 2) its burst-error-detecting capability is $b=h$, i.e., all error bursts of length up to h bits are detected, and 3) its minimum distance is $d=4$, i.e., all double errors and all odd numbers of errors are detected. These properties are called the *guaranteed* error-detecting capability. The CRC may detect other errors, but not guaranteed, e.g., it can detect a large percentage of error bursts of length greater than h [2], [10], [15]. An important problem, which is NP-hard and is not addressed in this paper, is the determination of the undetected error probability of a code [7].

General-purpose computers and compilers are increasingly faster and more sophisticated. Software algorithms are commonly used in operations, modeling, simulations, and performance analysis of systems and networks. CRC implementation in software is desirable, because many computers do not have hardware circuits dedicated for CRC computation. However, software implementation of typical CRCs is slow, because it relies on many steps of the polynomial division during CRC computation. It is this speed limitation of CRCs that leads to the use of checksums (which are fast and typically do not rely on table lookup) as alternatives to CRCs in many high-speed networking applications, although checksums are weaker than CRCs. For example, the 16-bit ones-complement checksum is used in Internet protocol and the Fletcher checksum is used in ISO [5], [17]. There are also other fast error-detection codes [3], [4], [12], [13], but they do not have all the desirable properties of CRCs.

A relevant question is whether there is a new family of CRCs that are faster than the existing CRCs. In this paper, we introduce such CRCs, as well as a technique for their efficient implementation. For these fast CRCs, it is possible either to eliminate or to greatly reduce many steps of the polynomial division during their computation.

A common existing technique for reducing the many steps during CRC computation is to use table lookup, which requires extra memory [9], [14], [15], [16]. In contrast, even without table lookup, our fast CRCs require only a small number of steps for their computation. Algorithms that do not rely on table lookup have an advantage of being less dependent on issues such as cache architecture and cache miss. In particular, it is possible to use as low as 1.5 operations per input message byte to encode our fast 64-bit CRC (which is implemented in C and requires no table lookup).

The paper is organized as follows: In Section 2, we review known facts about CRCs, which serve as the background for our discussions. We present several different algorithms for computing CRCs, some of which are designed especially for our fast CRCs. In Section 3, we identify the form of the generator polynomials for the fast CRCs, and introduce a new technique for their implementation. We then determine their *guaranteed* error-detecting capability: the minimum distance, the burst-error-detecting capability, and the maximum code length. In Section 4, we discuss CRC software complexity and show that our fast CRCs are typically faster than other CRCs. In Section 5, we present summaries and extensions of the paper.

1.1 Notation and Convention

In this paper, we consider polynomials that have binary coefficients 0 and 1. Thus, all polynomial operations are performed in the binary field GF(2), i.e., by using polynomial arithmetic modulo 2. Let $A(X)$ and $M(X)$ be two polynomials, then $\mathrm{R}_{M(X)}[A(X)]$ denotes the remainder

- *G.D. Nguyen is with the Information Technology Division, Naval Research Laboratory, Washington, DC 20375.*
E-mail: gam.nguyen@nrl.navy.mil.

Manuscript received 7 Mar. 2008; revised 8 Oct. 2008; accepted 11 Mar. 2009; published online 15 June 2009
Recommended for acceptance by C. Bolchini.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-03-0105.
Digital Object Identifier no. 10.1109/TC.2009.83.

0018-9340/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

polynomial that is obtained when $A(X)$ is divided by $M(X)$. We must have $\mathrm{degree}(\mathrm{R}_{M(X)}[A(X)]) < \mathrm{degree}(M(X))$.

An s-tuple denotes a block of s bits $A = (a_{s-1}, a_{s-2}, \ldots, a_1, a_0)$, which is also presented by the binary polynomial $a_{s-1}X^{s-1} + a_{s-2}X^{s-2} + \cdots + a_1X + a_0$ of degree less than s. We use the closely related notation $A(X)$ to denote this polynomial, i.e., A is composed of the binary coefficients of $A(X)$. Thus, the tuple A and the polynomial $A(X)$ are equivalent and can be used interchangeably. Typically, the polynomial notation is used to describe the mathematical properties of codes, whereas the tuple notation is used to describe the algorithmic properties (such as pseudocodes and computer programs) of codes. If $Q_1(X)$ and $Q_2(X)$ are s_1-tuple and s_2-tuple, respectively, then the $(s_1 + s_2)$-tuple $(Q_1(X), Q_2(X))$ denotes the polynomial $Q_1(X)X^{s_2} + Q_2(X)$, which is the concatenation of $Q_2(X)$ to $Q_1(X)$.

In this paper, we are interested in CRCs that have low software complexity. *Software* complexity of an algorithm refers to the number of operations (i.e., operation count) used to implement the algorithm (whereas *hardware* complexity refers to the number of gates used to implement the algorithm). Suppose that we have two CRCs that operate under similar environments and use similar types of operations, but one CRC requires lower operation count (e.g., having a smaller loop) than the other. It is likely that the CRC with lower operation count (i.e., lower software complexity) will result in faster encoding. Thus, complexity correlates with speed. However, the amount of the correlation also depends on many other complicating factors such as memory speed, cache size, compiler, operating system, pipelining, and CPU architecture. A CRC is called "fast" if it has low software complexity and low memory requirement (e.g., it requires no lookup table or only a small lookup table). A CRC is called "faster" than another if, for a similar level of memory requirement, it has lower software complexity.

An algorithm (or implementation) is called *bitwise* if it does not use table lookup. Note that a bitwise algorithm does not necessarily involve only bit-by-bit manipulation or computation. Fast checksums are typically bitwise. Bitwise algorithms, which do not rely on table lookup, have an advantage of being less dependent on issues such as cache architecture, cache miss, and software code space. Ideally, fast CRC algorithms should have low complexity and be bitwise. Thus, unless explicitly stated, we focus on bitwise algorithms in this paper. Table-lookup algorithms are presented in [19, Appendix A] which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2009.83.

The notation (k, l, d) denotes a systematic code with $k =$ the total bit length of the code, $l =$ the bit length of the input message, and $d =$ the minimum distance of the code. The burst-error-detecting capability of a code is denoted by b. To facilitate cross-references, we label some blocks of text as "Remarks," which are an integral part of the presentation and should not be viewed as isolated observations or comments.

2 CRC Algorithms

In this section, we review some known facts about software CRC implementation (e.g., see [2], [4], [6], [9], [12], [14], [15], [16]). To lay a firm foundation for our later discussions, we present these facts in more precise and general forms than those often seen in the literature. Our presentation is a straightforward generalization of the results in [15].

2.1 General CRC Theory

Suppose that we use an h-bit CRC, generated by a polynomial $M(X)$ of degree h, to protect an input message $U(X)$, which has l bits. By definition, the check polynomial $P(X)$ is the remainder that is obtained by dividing $U(X)X^h$ by $M(X)$, i.e., $P(X) = \mathrm{R}_{M(X)}\left[U(X)X^h\right]$. Because computers can process tuples of bits (e.g., bytes or words) at a time, codes having efficient software implementation should be encoded on tuples. Typical modern processors can efficiently handle tuples of 8, 16, 32, and 64 bits.

Let $s > 0$ be any positive integer. We can write $l = r + (n-1)s$, for some $n > 0$ and $0 < r \leq s$. We then process the CRC by dividing the input message $U(X)$ into n tuples. The first tuple has r bits, and all the other tuples have s bits. Because $r \leq s$, we can then insert $(s - r)$ zeros to the left of $U(X)$ to increase its length from l to $l' = l + s - r = ns$, without affecting the CRC computation, because $\mathrm{R}_{M(X)}[(0, 0, \ldots, 0, U(X))X^h] = \mathrm{R}_{M(X)}[U(X)X^h] = P(X)$. That is, the first tuple now also has s bits, the $(s - r)$ left-hand bits of which are always zeros.

Because each tuple i has s bits, it can be represented by a polynomial $Q_i(X)$ of degree $< s$. Thus, the input message is represented by $U(X) = (Q_0(X), Q_1(X), \ldots, Q_{n-1}(X))$. We emphasize that, for given h and l, we are free to choose the value of s (commonly chosen values are $s = 8$, 16, 32, and 64 bits). As shown later, the choice of s can have significant impact on CRC speed.

Define $U_i(X) = (Q_0(X), Q_1(X), \ldots, Q_i(X))$ to be the first $i + 1$ input tuples, i.e.,

$$\begin{aligned} U_0(X) &= Q_0(X) \\ U_1(X) &= (Q_0(X), Q_1(X)) \\ &\cdots \\ U_{n-1}(X) &= (Q_0(X), Q_1(X), \ldots, Q_{n-1}(X)) \\ &= U(X). \end{aligned}$$

Thus, for $i = 1, 2 \ldots, n-1, U_i(X)$ is determined from $U_{i-1}(X)$ and $Q_i(X)$ by

$$\begin{aligned} U_i(X) &= (U_{i-1}(X), Q_i(X)) \\ &= U_{i-1}(X)X^s + Q_i(X). \end{aligned} \tag{2}$$

For $i = 0, 1, \ldots, n-1$, let $P_i(X)$ be the CRC check polynomial for the partial input message $U_i(X)$, i.e.,

$$P_i(X) = \mathrm{R}_{M(X)}\left[U_i(X)X^h\right]. \tag{3}$$

In particular, we have $P_0(X) = \mathrm{R}_{M(X)}\left[Q_0(X)X^h\right]$, and

$$\begin{aligned} P_{n-1}(X) &= \mathrm{R}_{M(X)}\left[U_{n-1}(X)X^h\right] \\ &= \mathrm{R}_{M(X)}\left[U(X)X^h\right] \\ &= P(X), \end{aligned}$$

which is the CRC check polynomial for the entire input message $U(X)$.

```
$B = 0$;
for $(0 \leq i < n)$
    {
    $A = B + Q_i X^{h-s}$;
    $B = \mathrm{R}_M [AX^s]$;
    }
$P = B$;
return $P$;
```

Fig. 1. CRC Algorithm 1 for computing the check h-tuple P from the input s-tuples $Q_0, \ldots, Q_{n-1}$ ($s < h$).

Substituting (2) into (3), we have

$$\begin{aligned} P_i(X) &= \mathrm{R}_{M(X)}\left[U_i(X)X^h\right] \\ &= \mathrm{R}_{M(X)}\left[(U_{i-1}(X)X^s + Q_i(X))X^h\right] \\ &= \mathrm{R}_{M(X)}\left[(U_{i-1}(X)X^h)X^s\right] + \mathrm{R}_{M(X)}\left[Q_i(X)X^h\right]. \end{aligned}$$

Using (3), we then have

$$\begin{aligned} P_i(X) &= \mathrm{R}_{M(X)}[P_{i-1}(X)X^s] + \mathrm{R}_{M(X)}\left[Q_i(X)X^h\right] \\ &= \mathrm{R}_{M(X)}\left[P_{i-1}(X)X^s + Q_i(X)X^h\right], \end{aligned} \tag{4}$$

for $i = 1, 2, \ldots, n-1$. Note that (4) is a straightforward generalization of a result in [15], which deals with the special cases $h = 16$ and $s \in \{8, 16\}$. Thus, the check tuple $P_i(X)$ is computed from $Q_i(X)$ and the previous check tuple $P_{i-1}(X)$. Recall that $P_0(X) = \mathrm{R}_{M(X)}\left[Q_0(X)X^h\right]$ and $P(X) = P_{n-1}(X)$ is the CRC check tuple for $U(X)$. Using (4), $P(X)$ is then computed via the following pseudocode:

```
$P = 0$;
for $(0 \leq i < n)$
    $P = \mathrm{R}_M \left[PX^s + Q_i X^h\right]$;
return $P$;
```

Remark 1. We now review the computational complexity of polynomial division, which is needed in CRC computation. Given two polynomials $W(X)$ and $Y(X)$, let $V(X) = \mathrm{R}_{W(X)}[Y(X)]$ be the remainder polynomial that is obtained when $Y(X)$ is divided by $W(X)$. Let w and y be the degrees of $W(X)$ and $Y(X)$, respectively. If $y < w$ (i.e., $y - w + 1 \leq 0$), then $V(X) = Y(X)$, i.e., no polynomial division is needed to obtain the remainder $V(X)$. If $y \geq w$, we then need a polynomial division that requires a loop of $y - w + 1$ iterations to obtain the remainder $V(X)$ (see [8, p. 421]). To summarize, the polynomial "long division" for computing $\mathrm{R}_{W(X)}[Y(X)]$ requires a loop of $\max(0, y - w + 1)$ iterations.

2.2 Two CRC Algorithms

From (4), we have

$$P_i(X) = \mathrm{R}_{M(X)}\left[(P_{i-1}(X) + Q_i(X)X^{h-s})X^s\right] \tag{5}$$

if $s < h$, and

$$P_i(X) = \mathrm{R}_{M(X)}\left[(P_{i-1}(X)X^{s-h} + Q_i(X))X^h\right] \tag{6}$$

if $s \geq h$. The CRC algorithms based on (5) and (6), called Algorithms 1 and 2, are shown in Figs. 1 and 2, respectively.

```
$B = 0$;
for $(0 \leq i < n)$
    {
    $A = BX^{s-h} + Q_i$;
    $B = \mathrm{R}_M \left[AX^h\right]$;
    }
$P = B$;
return $P$;
```

Fig. 2. CRC Algorithm 2 for computing the check h-tuple P from the input s-tuples $Q_0, \ldots, Q_{n-1}$ ($s \geq h$).

2.3 Two Alternative CRC Algorithms

We now present two alternative CRC algorithms, which will be applied to our fast CRCs (see Section 3).

Case 1: $s < h$. The CRC check polynomial $P_j(X)$ for the partial input message $U_j(X)$ can be split into two parts as

$$P_j(X) = (P_{j,1}(X), P_{j,2}(X)) = P_{j,1}(X)X^{h-s} + P_{j,2}(X), \tag{7}$$

where $P_{j,1}(X)$ and $P_{j,2}(X)$ are polynomials with $\mathrm{degree}(P_{j,1}(X)) < s$ and $\mathrm{degree}(P_{j,2}(X)) < h - s$. That is, $P_{j,1}(X)$ and $P_{j,2}(X)$ are the s left-hand bits and $(h - s)$ right-hand bits of $P_j(X)$, respectively. Substituting (7) into (4), we have

$$\begin{aligned} P_i(X) &= \mathrm{R}_{M(X)}\left[(P_{i-1,1}(X)X^{h-s} + P_{i-1,2}(X))X^s\right] \\ &\quad + \mathrm{R}_{M(X)}\left[Q_i(X)X^h\right] \\ &= \mathrm{R}_{M(X)}\left[\left(P_{i-1,1}(X) + Q_i(X)\right)X^h\right] \\ &\quad + \mathrm{R}_{M(X)}\left[P_{i-1,2}(X)X^s\right]. \end{aligned}$$

Because $\mathrm{degree}(P_{i-1,2}(X)X^s) < h = \mathrm{degree}(M(X))$, we have $\mathrm{R}_{M(X)}[P_{i-1,2}(X)X^s] = P_{i-1,2}(X)X^s$. Thus,

$$P_i(X) = \mathrm{R}_{M(X)}\left[(P_{i-1,1}(X) + Q_i(X))X^h\right] + P_{i-1,2}(X)X^s. \tag{8}$$

The CRC algorithm based on (8), called Algorithm 3, is shown in Fig. 3.

Case 2: $s \geq h$. Multiplying both sides of (6) by X^{s-h}, we have

$$\begin{aligned} P_i(X)X^{s-h} &= \left(\mathrm{R}_{M(X)}\left[(P_{i-1}(X)X^{s-h} + Q_i(X))X^h\right]\right)X^{s-h} \\ &= \mathrm{R}_{M(X)X^{s-h}}\left[(P_{i-1}(X)X^{s-h} + Q_i(X))X^h X^{s-h}\right] \\ &= \mathrm{R}_{M(X)X^{s-h}}\left[(P_{i-1}(X)X^{s-h} + Q_i(X))X^s\right]. \end{aligned} \tag{9}$$

```
$P = 0$;
for $(0 \leq i < n)$
    {
    $P_1 = s$ left-hand bits of $P$;
    $P_2 = (h - s)$ right-hand bits of $P$;
    $A = P_1 + Q_i$;
    $B = \mathrm{R}_M \left[AX^h\right]$;
    $P = B + P_2 X^s$;
    }
return $P$;
```

Fig. 3. CRC Algorithm 3 for computing the check h-tuple P from the input s-tuples $Q_0, \ldots, Q_{n-1}$ ($s < h$).

```
1 $B = 0;$
2 for $(0 \leq i < n)$
3     {
4     $A = B + Q_i;$
5     $B = \mathrm{R}_N\left[AX^s\right];$
6     }
7 $P = h$ left-hand bits of $B$;
8 return $P$;
```

Fig. 4. CRC Algorithm 4 for computing the check h-tuple P from the input s-tuples $Q_0, \ldots, Q_{n-1}$ $(s \geq h)$.

Define $L_j(X) = P_j(X)X^{s-h}$. From (9), we then have

$$L_i(X) = \mathrm{R}_{N(X)}[(L_{i-1}(X) + Q_i(X))X^s], \qquad (10)$$

where $N(X) = M(X)X^{s-h}$. Thus, $L_i(X)$ is computed from $L_{i-1}(X)$ and $Q_i(X)$.

Note that $L_0(X) = P_0(X)X^{s-h}$, where $P_0(X) = \mathrm{R}_{M(X)}\left[Q_0(X)X^h\right]$. We then have

$$\begin{aligned} L_0(X) &= (\mathrm{R}_{M(X)}\left[Q_0(X)X^h\right])X^{s-h} \\ &= \mathrm{R}_{M(X)X^{s-h}}\left[Q_0(X)X^hX^{s-h}\right] \\ &= \mathrm{R}_{N(X)}[Q_0(X)X^s]. \end{aligned}$$

Because $L_i(X) = P_i(X)X^{s-h}$, the term $P_i(X)$ is obtained by shifting $L_i(X)$ to the right by $(s-h)$ bits. Note that $\mathrm{degree}(L_i(X)) < s$. We will show in Remark 2 that computing $P_i(X)$ via (10) is slightly faster than via (6). The CRC algorithm based on (10), called Algorithm 4, is shown in Fig. 4.

Remark 2. Suppose that $s \geq h$. The check polynomial $P(X) = P_{n-1}(X) = \mathrm{R}_{M(X)}[U_{n-1}(X)X^h]$ can then be computed by Algorithm 2 (Fig. 2) or by Algorithm 4 (Fig. 4). We now show that, for bitwise implementation, Algorithm 4 is slightly faster than Algorithm 2. By comparing these two algorithms, we observe the following. First, the computation of $\mathrm{R}_{M(X)}[A(X)X^h]$ (Fig. 2) and the computation of $\mathrm{R}_{N(X)}[A(X)X^s]$ (Fig. 4) have the same complexity, because each requires s iterations (by Remark 1). Next, the factor X^{s-h} at line 4 of Fig. 2 disappears from line 4 of Fig. 4. Finally, one extra operation is required at line 7 of Fig. 4 to extract the h left-hand bits of the final $B(X)$. The above observations imply that Algorithm 4 requires $n-1$ fewer operations than Algorithm 2. Thus, for bitwise implementation, we will use Algorithm 4 when $s \geq h$.

2.4 Basic CRC Algorithms

Given an input message $U(X)$ and a generator polynomial $M(X)$ of degree h, Algorithms 1-4 produce the same CRC check tuple $P(X)$. That is, they are four different ways for accomplishing the same thing. The main difference among these algorithms is how the input message is divided into s-tuples $Q_i(X)$. Algorithms 1 and 3 are for $s < h$, whereas Algorithms 2 and 4 are for $s \geq h$. As shown later, CRC speed depends on the choice of s. For flexibility, we allow the possibility that the same CRC is used by computers that have different architectures and capabilities. For example, one computer can choose a value of s for encoding a message to transmit to another computer (with different capabilities), which can choose a different value of s for detecting the errors in the received message.

The above CRC algorithms require polynomial divisions. In particular, Algorithm 1 requires the polynomial division $\mathrm{R}_{M(X)}[A(X)X^s]$, Algorithms 2 and 3 require the polynomial division $\mathrm{R}_{M(X)}[A(X)X^h]$, and Algorithm 4 requires the polynomial division $\mathrm{R}_{N(X)}[A(X)X^s]$. To simplify the presentation, we will use the single notation $B(X)$ to denote all these polynomial divisions, i.e., we define

$$B(X) = \begin{cases} \mathrm{R}_{M(X)}[A(X)X^s] & \text{(Algorithm 1)}, \\ \mathrm{R}_{M(X)}[A(X)X^h] & \text{(Algorithms 2 and 3)}, \\ \mathrm{R}_{N(X)}[A(X)X^s] & \text{(Algorithm 4)}, \end{cases} \qquad (11)$$

where $N(X) = M(X)X^{s-h}$. Note that $\mathrm{degree}(A(X)) < h$ in Algorithm 1, and $\mathrm{degree}(A(X)) < s$ in Algorithms 2-4. As seen in Figs. 1-4, CRC computation using any of the above four algorithms requires the computation of $B(X)$ for n times.

A known technique for computing $B(X)$ is to use the polynomial long division algorithm mentioned in Remark 1. For example, consider Algorithms 2 and 3. We then have $B(X) = \mathrm{R}_{M(X)}[A(X)X^h]$, where $\mathrm{degree}(A(X)) < s$. Because $\mathrm{degree}(A(X)X^h) \leq s + h - 1$ and $\mathrm{degree}(M(X)) = h$, by Remark 1, $B(X)$ can be computed via the polynomial long division that requires a loop of s iterations. Similarly, it can be shown that computing $B(X)$ in Algorithms 1 and 4 also requires a loop of s iterations. That is, the computational complexity for computing $B(X)$ is $O(s)$.

Definition 1. *The technique for computing the polynomial $B(X)$ as given in (11) is called the "basic" technique. Using the polynomial long division, $B(X)$ can be computed in s iterations. An algorithm (or a CRC) is basic if it uses the basic technique for computing $B(X)$.*

3 Fast CRCs

Recall that we are given an input message $U_{n-1}(X) = (Q_0(X), Q_1(X), \ldots, Q_{n-1}(X))$, where $Q_i(X)$ is an s-tuple. We protect this message by an h-bit CRC generated by a polynomial $M(X)$ of degree h. The check h-tuple

$$P(X) = P_{n-1}(X) = \mathrm{R}_{M(X)}\left[U_{n-1}(X)X^h\right]$$

can be computed by Algorithm 1 or 3 (if $s < h$), or by Algorithm 2 or 4 (if $s \geq h$). We emphasize that each of these algorithms requires the calculation of $B(X)$ defined in (11), which involves the polynomial division.

3.1 Fast h-Bit CRCs

Our goal is to find some CRCs that have fast implementation, i.e., to find a new family of generator polynomials $M(X)$ for CRCs that have low complexity. Recall that the CRC algorithms (Figs. 1-4) depend on the term $B(X)$. Computation of $B(X)$ is also the most expensive step in the algorithms. Thus, finding fast CRCs requires finding the polynomials $M(X)$ that yield fast computation of $B(X)$.

The first technique for computing $B(X)$ is the *basic* technique in Definition 1. Using the polynomial division, we can compute $B(X)$ by a loop of s iterations. In the following, we present the second technique, called the *new* technique, for computing $B(X)$. While the new technique is applicable to any generator polynomial $M(X)$, it is more effective for some special CRC generator polynomials, called the *fast* polynomials. Recall that the basic CRCs can use Algorithm 1 or 3 (if $s < h$), or by Algorithm 2 or 4 (if $s \geq h$). However, as seen in the following, the fast CRCs use only Algorithms 3 (for $s < h$) and 4 (for $s \geq h$) for their bitwise implementation.

We now introduce a new family of CRCs, which are generated by the following polynomials

$$F_h(X) = X^h + X^2 + X + 1, \tag{12}$$

for all $h \geq 4$. We ignore the case $h = 3$, which yields the trivial repetition code $\{(0000), (1111)\}$. We call $F_h(X)$ the "fast polynomial," which can be factored into

$$X^h + X^2 + X + 1 = (X + 1)G_{h-1}(X),$$

where

$$G_m(X) = X^m + X^{m-1} + \cdots + X^3 + X^2 + 1, \tag{13}$$

i.e., $G_m(X)$ includes all the terms except X. At first, it is not clear why this particular polynomial $F_h(X)$ will speed up the computation of $B(X)$. We now introduce a technique that is applied to $F_h(X)$ to yield fast computation of $B(X)$.

By considering Algorithms 3 and 4, we have from (11)

$$B(X) = \begin{cases} \mathrm{R}_{M(X)}\left[A(X)X^h\right] & \text{if } s < h, \\ \mathrm{R}_{N(X)}[A(X)X^s] & \text{if } s \geq h, \end{cases} \tag{14}$$

where $N(X) = M(X)X^{s-h}$, and $A(X)$ is a polynomial of degree less than s. We now transform $B(X)$ into a new form that will be used by the fast CRCs. First, note that

$$\begin{aligned} &\mathrm{R}_{M(X)}\left[A(X)(X^h + M(X))\right] \\ &\quad = \mathrm{R}_{M(X)}\left[A(X)X^h\right] + \mathrm{R}_{M(X)}[A(X)M(X)] \\ &\quad = \mathrm{R}_{M(X)}\left[A(X)X^h\right] \end{aligned}$$

because $\mathrm{R}_{M(X)}[A(X)M(X)] = 0$. Similarly, we have

$$\mathrm{R}_{N(X)}[A(X)(X^s + N(X))] = \mathrm{R}_{N(X)}[A(X)X^s].$$

Thus, (14) becomes

$$B(X) = \begin{cases} \mathrm{R}_{M(X)}\left[A(X)(X^h + M(X))\right] & \text{if } s < h, \\ \mathrm{R}_{N(X)}[A(X)(X^s + N(X))] & \text{if } s \geq h, \end{cases} \tag{15}$$

where $N(X) = M(X)X^{s-h}$.

Definition 2. *Using Algorithms 3 and 4, the technique (15) for computing the polynomial $B(X)$ is called the "new" technique. The CRC that is generated by the fast polynomial $F_h(X) = X^h + X^2 + X + 1$ and uses the new technique for computing $B(X)$ is called the fast h-bit CRC.*

Theorem 1. *Using Algorithms 3 and 4, the polynomial $B(X)$ for the fast CRC generated by $F_h(X) = X^h + X^2 + X + 1$ is given by*

$$B(X) = \begin{cases} \mathrm{R}_{F_h(X)}[A(X)(X^2 + X + 1)] & \text{if } s < h, \\ \mathrm{R}_{N(X)}\left[A(X)X^{s-h}(X^2 + X + 1)\right] & \text{if } s \geq h, \end{cases} \tag{16}$$

where $N(X) = F_h(X)X^{s-h}$, and $A(X)$ is a polynomial of degree less than s. Further, using the polynomial division, $B(X)$ can be computed with $\max(0, s - h + 2)$ *iterations.*

Proof. Relation (16) follows by using (15) with $M(X) = F_h(X)$ and $N(X) = F_h(X)X^{s-h}$. First, suppose that $s < h$. Then, $B(X) = \mathrm{R}_{F_h(X)}[A(X)(X^2 + X + 1)]$. Because $\mathrm{degree}(F_h(X)) = h$ and $\mathrm{degree}(A(X)(X^2 + X + 1)) < s + 2$, from Remark 1, $B(X)$ can be computed with $\max(0, s - h + 2)$ iterations. Next, suppose that $s \geq h$. Then, $B(X) = \mathrm{R}_{N(X)}\left[A(X)X^{s-h}(X^2 + X + 1)\right]$. Because $\mathrm{degree}(N(X)) = s$ and $\mathrm{degree}(A(X)X^{s-h}(X^2 + X + 1)) < 2s - h + 2$, Remark 1 implies that $B(X)$ can also be computed with $\max(0, s - h + 2)$ iterations. □

Let us briefly compare the computational complexity of $B(X)$ for 1) the basic h-bit CRC generated by $M(X)$ and 2) the fast h-bit CRC generated by $F_h(X)$. For the basic CRC, by Definition 1, $B(X)$ is computed via a loop of s iterations, regardless of the form of $M(X)$. However, for the fast CRC, by Theorem 1, $B(X)$ is computed via a loop of only $\max(0, s - h + 2)$ iterations. Thus, the fast CRC is much faster than the basic CRC if s is chosen such that $s - h + 2$ is much small than s. Further, if $s + 1 < h$, then $\max(0, s - h + 2) = 0$ and $B(X) = A(X)(X^2 + X + 1)$, i.e., the polynomial division is eliminated. Section 4 presents CRC software complexity in more detail.

We emphasize that *fast* CRC denotes a CRC that meets the following two conditions: 1) the CRC is generated by the *fast* polynomial $F_h(X) = X^h + X^2 + X + 1$, and 2) the polynomial $B(X)$ is computed via Theorem 1 by applying the *new* technique (15) to $F_h(X)$. That is, fast CRC refers to a CRC that is generated by a specific polynomial and is implemented by a specific technique. Note that a CRC that meets only one of the above two conditions may not have any speed advantage over a basic CRC. For example, suppose that, instead of the new technique (15), the basic technique (in Definition 1) is applied to the CRC generated by the fast polynomial $F_h(X)$. This CRC is then not different from a basic CRC in terms of computational complexity. Application of the new technique to polynomials other than $F_h(X)$ is considered in [19, Appendix C].

To summarize, the fast h-bit CRC is generated by $F_h(X) = X^h + X^2 + X + 1$. Under bitwise implementation, the fast CRC uses Algorithm 3 if $s < h$ and Algorithm 4 if $s \geq h$. The term $B(X)$ in these algorithms is given in Theorem 1.

3.2 A Fast 16-Bit CRC

We now consider the important case $h = 16$. Many CRCs (as well as weaker checksums) used in practice have 16 check bits, e.g., the CRC-16 and CRC-CCITT mentioned in Section 1. With a small amount of overhead, these CRCs can have length up to $2^{15} - 1$ bits $\approx 4{,}096$ bytes. Our goal here is to present a concrete example of a new 16-bit CRC that is not only much faster than but also as good as existing 16-bit CRCs.

Our new 16-bit CRC is generated by

$$F_{16}(X) = X^{16} + X^2 + X + 1, \tag{17}$$

which can be factored into

$$F_{16}(X) = (X + 1)G_{15}(X),$$

where

$$G_{15}(X) = X^{15} + X^{14} + \cdots + X^3 + X^2 + 1.$$

It can be shown that $G_{15}(X)$ is a primitive polynomial, i.e., $F_{16}(X)$ is a product of $X+1$ and a primitive polynomial (however, as seen later, this is not true for many values of h). Thus, this fast 16-bit CRC also has length up to $2^{15}-1$ bits. Although the polynomial (17) is different from the generator polynomials for existing 16-bit CRCs, it does generate a CRC that has the same guaranteed error-detecting capability as existing 16-bit CRCs. From Theorem 1, we have

$$B(X) = \begin{cases} \mathrm{R}_{F_{16}(X)}[A(X)(X^2+X+1)] & \text{if } s < 16, \\ \mathrm{R}_{N(X)}[A(X)X^{s-16}(X^2+X+1)] & \text{if } s \geq 16, \end{cases}$$

where $N(X) = F_{16}(X)X^{s-16}$.

In the following, we consider two cases: $s=8$ and $s=16$. First, assume that $s=8$, i.e., the input message is organized in 8-bit bytes. Because $s<16$, we have $B(X) = \mathrm{R}_{F_{16}(X)}[A(X)(X^2+X+1)]$. Because $\mathrm{degree}(A(X)) < s = 8$, we have $\mathrm{degree}(A(X)(X^2+X+1)) < 10$, which is smaller than $\mathrm{degree}(F_{16}(X)) = 16$. From Remark 1, we have

$$\begin{aligned} B(X) &= A(X)(X^2+X+1) \\ &= A(X)X^2 + A(X)X + A(X), \end{aligned}$$

i.e., $B(X)$ is simply the sum of $A(X)$ and its translations. Thus, computing $B(X)$ via the new technique requires no polynomial division. In contrast, computing $B(X)$ via the basic technique requires the polynomial division that has a loop of $s=8$ iterations (see Definition 1).

Next, assume that $s=h=16$, i.e., the input message is organized in 16-tuples. Because $s=16$, we have $\mathrm{degree}(A(X)) < 16$ and $\mathrm{degree}(A(X)(X^2+X+1)) < 18$. Thus, by Remark 1, $B(X)$ is computed by polynomial division that has a loop of two iterations. This contrasts with computing $B(X)$ via the basic technique, which requires a loop of $s=16$ iterations (see Definition 1). Thus, the loop iteration count of our new technique is less than that of the basic technique by the factor of $16/2 = 8$.

To summarize, when the input message is organized in s-tuples, it is possible to have a fast 16-bit CRC that requires no polynomial division (when $s=8$), or that requires the polynomial division that has only two loop iterations (when $s=16$). Further, this fast 16-bit CRC has the same guaranteed error-detecting capability as existing 16-bit CRCs.

When computing $B(X)$ via the new technique, although the case $s=16$ requires more loop iterations than the case $s=8$, we will see later in Section 4 that the case $s=16$ has lower *overall* computational complexity (i.e., lower overall operation count per input byte). This is because, when $s=16$, there is no need to compute $P_{j,1}(X)$ and $P_{j,2}(X)$ as defined in (7). Further, the overhead processing cost per input byte when $s=16$ is lower than when $s=8$. The C programs for the fast 16-bit CRC are shown in Fig. 8 and in [19, Appendix A, Fig. 12].

3.3 Error-Detection Capability of Fast CRCs

Recall that the h-bit CRC generated by $M(X)$ given in (1) has minimum distance $d=4$ if its total bit length $\leq 2^{h-1}-1$. We define the *maximum length* of an error-detection code to be the total bit length at or below which its minimum distance is $d \geq 3$, i.e., beyond which its minimum distance will reduce to $d=2$. Thus, the maximum length of the h-bit CRC generated by (1) is $2^{h-1}-1$ bits. In the following, we derive the maximum length of the fast CRCs, which is based on polynomial periodicity.

h	period	$\frac{2^{h-1}-1}{\text{period}}$
4	7	1
5	14	1.07143
6	31	1
7	60	1.05
8	127	1
9	254	1.00394
10	465	1.09892
11	868	1.17857
12	595	3.44034
13	4094	1.00024
14	8191	1
15	3276	5.00092
16	32767	1
17	9362	7.00011
18	38227	3.42875
19	229348	1.14299
20	516033	1.016
21	1048574	1
22	126945	16.5202
23	803148	5.22233
24	8388607	1
25	917490	18.286
26	584073	57.449
27	65011588	1.03226
28	87381	1536.01
29	268435454	1
30	5013351	107.088
31	1900428	565
32	2097151	1024
33	4194302	1024
34	408944445	21.0051
35	5637144492	3.04762
36	270532479	127.008
37	2.080×10^{10}	3.30323
38	$2^{37}-1$	1
39	4831838172	56
40	3.006×10^{10}	18.2857
48	$2^{47}-1$	1
56	3.573×10^{16}	1.00837
64	$2^{63}-1$	1
128	$2^{127}-1$	1

Fig. 5. The period of $F_h(X) = X^h + X^2 + X + 1$, which equals the maximum length of the fast h-bit CRC generated by $F_h(X)$.

By definition, the *period* of a polynomial $G(X)$ is the smallest positive integer i such that $\mathrm{R}_{G(X)}[X^i] = 1$. In particular, if $G(X)$ is the product of $X+1$ and a primitive polynomial of degree $h-1$, then it can be shown that $G(X)$ has period $2^{h-1}-1$. Note that some polynomials, such as X^2, do not have periods.

The period of the fast polynomial $F_h(X) = X^h + X^2 + X + 1$ can be computed directly from definition (for small h) or from the technique in [1, Section 6.2]. The periods of $F_h(X), h \geq 4$, are shown in Fig. 5. The following theorems, which are slight variations of well-known results from cyclic codes [10, Chapter 4], show that the maximum length of a CRC equals the period of its generator polynomial.

Theorem 2. *Let C be a CRC generated by a polynomial $M(X)$ of degree $h \geq 3$. Assume that X is not a factor of $M(X)$. Let n_{b}*

and d be the bit length and minimum distance of C, respectively. We then have

1. *$d \geq 3$ if $n_{\mathrm{b}} \leq$ period of $M(X)$.*
2. *$d = 2$ if $n_{\mathrm{b}} >$ period of $M(X)$.*
3. *C detects all error bursts of length up to h bits, i.e., $b = h$.*

Proof. Let t be the period of $M(X)$. We must have $t \geq h$. By definition, each codeword of C has the form

$$V(X) = U(X)X^h + P(X),$$

where $U(X)$ is the polynomial representing the input message, and $P(X)$ is the check polynomial. Because $P(X) = \mathrm{R}_{M(X)}\left[U(X)X^h\right]$, we have

$$U(X)X^h = K(X)M(X) + P(X),$$

for some polynomial $K(X)$. Thus, we have

$$V(X) = U(X)X^h + P(X) = K(X)M(X),$$

i.e., C is a linear code. If $d = 1$, then $X^i = K(X)M(X)$, for some i. This implies that $M(X) = X^j$ for some j, which contradicts our assumption that X is not a factor of $M(X)$. Thus, $d \geq 2$.

1. We now prove, by contradiction, the statement $d \geq 3$ if $n_{\mathrm{b}} \leq$ period of $M(X)$. Thus, suppose that there is a codeword $V(X)$ with length $n_{\mathrm{b}} \leq t$ and weight 2. Then, $V(X) = X^j + X^i$ for some i and j such that $n_{\mathrm{b}} > j > i \geq 0$. Thus, $V(X) = X^i(X^{j-i} + 1)$.

   We also have $V(X) = K(X)M(X)$ for some polynomial $K(X)$. Because X is not a factor of $M(X)$ by assumption, $M(X)$ must divide $X^{j-i} + 1$, i.e., $\mathrm{R}_{M(X)}[X^{j-i}] = 1$. Thus, $j - i \geq t =$ period of $M(X)$. Then, $j \geq t \geq n_{\mathrm{b}}$, which contradicts the condition $n_{\mathrm{b}} > j$. Thus, all the codewords of length $n_{\mathrm{b}} \leq t$ must have weight ≥ 3, i.e., $d \geq 3$.
2. We construct a codeword with length $> t$ and weight 2 as follows: Let $U(X) = X^{t-h}$. Then, $P(X) = \mathrm{R}_{M(X)}\left[U(X)X^h\right] = \mathrm{R}_{M(X)}[X^t]$. We have $P(X) = 1$ because t is the period of $M(X)$. Thus, the codeword $V(X) = U(X)X^h + P(X) = X^t + 1$ has length $t + 1$ and weight 2. That is, $d = 2$ if $n_{\mathrm{b}} > t$.
3. The fact that C detects all error bursts of length up to h bits (i.e., $b = h$) is well known [10]. □

Theorem 3. *Let C be the CRC generated by the fast polynomial $F_h(X) = X^h + X^2 + X + 1$. Let n_{b} and d be the bit length and minimum distance of C, respectively. We then have*

1. *$d = 4$ if $n_{\mathrm{b}} \leq$ period of $F_h(X)$.*
2. *$d = 2$ if $n_{\mathrm{b}} >$ period of $F_h(X)$.*
3. *C detects all error bursts of length up to h bits, i.e., $b = h$.*

Proof. Let t be the period of $F_h(X)$. From the proof of Theorem 2, every codeword of C has the form $V(X) = K(X)(X^h + X^2 + X + 1)$ for some polynomial $K(X)$. Thus, the codewords of C have even weight, i.e., d is even.

Suppose now that the input message is $U(X) = 1$. Then, $P(X) = \mathrm{R}_{F_h(X)}\left[X^h\right] = X^2 + X + 1$, which implies $V(X) = U(X)X^h + P(X) = X^h + X^2 + X + 1$. That is, the codeword $V(X)$ has weight 4. Thus, d is either 2 or 4. From Theorem 2.1, we must have $d = 4$ if $n_{\mathrm{b}} \leq t$. From Theorem 2.2, we must have $d = 2$ if $n_{\mathrm{b}} > t$. The fact that C detects all error bursts of length up to h bits is well known [10]. □

Recall that the maximum length of the h-bit CRC that is generated by $M(X)$ given in (1), which is the product of $X + 1$ and a primitive polynomial of degree $h - 1$, is $2^{h-1} - 1$. Fig. 5 shows that the maximum length of the fast h-bit CRC is also $2^{h-1} - 1$ in many important cases, namely when $h = 8, 16, 24, 48, 64, 128$. In fact, $F_h(X) = X^h + X^2 + X + 1$ is also the product of $X + 1$ and a primitive polynomial at these values of h, i.e., the polynomial $G_{h-1}(X)$ in (13) is primitive when $h = 8, 16, 24, 48, 64, 128$.

Fig. 5 also shows that the maximum lengths of many fast h-bit CRCs are substantially less than the upper bound $2^{h-1} - 1$ (e.g., when $h = 12$ and $h = 32$). However, in [19, Appendix C], we apply our new technique to more general generator polynomials to yield other fast CRCs whose maximum lengths can approach the upper bound.

4 CRC Software Complexity

We now analyze and compare CRC software complexity. *Software* complexity of an algorithm refers to the number of operations (i.e., operation count) used to implement the algorithm. Our goal in this paper is to compute the CRC check h-tuple $P(X)$ for an input message that consists of n tuples $Q_0(X), Q_1(X), \ldots, Q_{n-1}(X)$. Each tuple $Q_i(X)$ has s bits. This CRC can be either a basic CRC generated by a polynomial $M(X)$ of degree h, or the fast CRC generated by $F_h(X) = X^h + X^2 + X + 1$. For bitwise implementation, while Algorithm 1, 2, 3, or 4 can be used for the basic CRC, only Algorithm 3 or 4 are used for the fast CRC. The check tuple $P(X)$ is computed by using a loop that computes $B(X)$ for n times, where $B(X)$ is given in Definition 1 for the basic CRC and in Theorem 1 for the fast CRC.

In this section, we compute e_b and e_f, which denote the software operation counts per input byte required for computing the check tuple $P(X)$ for the basic CRC and the fast CRC, respectively. These operation counts will then be used to compare the complexity among our fast CRCs, the basic CRCs, the other fast CRCs in [4], and the block-parity checksum. An error-detection code is said to be "faster" than another if, for a similar level of memory requirement, it has lower software complexity.

4.1 General Complexity Analysis

We now provide the complexity analysis for the important case $s = h$ for the basic CRC and the fast CRC (other cases can be analyzed similarly). Both Algorithms 2 and 4 (shown in Figs. 2 and 4) then reduce to Fig. 6. Here, we have

$$B(X) = \mathrm{R}_{M(X)}[A(X)X^s] \tag{18}$$

for the basic CRC (see Definition 1), and

$$B(X) = \mathrm{R}_{F_h(X)}[A(X)(X^2 + X + 1)] \tag{19}$$

```
1 $B = 0$;
2 for $(0 \le i < n)$
3     {
4     $A = B + Q_i$;
5     $B = \begin{cases} \mathrm{R}_M\left[AX^s\right]; & \text{for basic CRC} \\ \mathrm{R}_F\left[A(X^2+X+1)\right]; & \text{for fast CRC} \end{cases}$
6     }
7 $P = B$;
8 return $P$;
```

Fig. 6. CRC algorithm ($s = h$).

for the fast CRC (see Theorem 1), where $A(X)$ is a polynomial of degree less than s. Note that different CRC algorithms refer to different techniques for computing $B(X)$. In particular, a CRC algorithm is called *table lookup* or *bitwise*, depending on whether the term $B(X)$ in the algorithm is computed with or without table lookup. The bitwise technique is presented in this section. The table-lookup technique is presented in [19, Appendix A].

Remark 3. The term $B(X) = \mathrm{R}_{M(X)}[A(X)X^s]$ in (18) can be computed as follows: First, we write $A(X)X^s = (\cdots(A(X)X)\cdots)X$. Thus, $B(X)$ can be computed in s iterations via the following pseudocode:

```
1 for $(0 \le j < s)$
2     $A = \mathrm{R}_M\,[AX]$;
3 $B = A$;
```

where $\mathrm{R}_M[AX]$ is computed by

$$\mathrm{R}_M[AX] = \begin{cases} AX + M & \text{if msb}(A) = 1, \\ AX & \text{if msb}(A) = 0, \end{cases} \tag{20}$$

where msb(A) denotes the most significant bit of A. The term $\mathrm{R}_M[AX]$ in (20) can also be computed by using a table $T[\,]$ of only two entries defined by $T[0] = 0$ and $T[1] = M$. We then have

$$\mathrm{R}_M[AX] = AX + T[\mathrm{msb}(A)]. \tag{21}$$

Let u be the operation count required for computing $\mathrm{R}_{M(X)}[A(X)X]$. Using Remark 3, the operation count required for computing $B(X)$ in (18) for the basic CRC is then $s(u + l_s)$, where l_s denotes the operation count for the loop overhead shown at line 1 of the pseudocode in Remark 3 (in particular, $l_s = 0$ if loop unrolling is used).

Let us now consider the term $B(X)$ in (19) for the fast CRC. We have

$$\begin{aligned} B(X) &= \mathrm{R}_{F_h(X)}\left[A(X)X^2\right] + \mathrm{R}_{F_h(X)}[A(X)X] + A(X) \\ &= \mathrm{R}_{F_h(X)}[B_1(X)X] + B_1(X) + A(X), \end{aligned} \tag{22}$$

where $B_1(X) = \mathrm{R}_{F_h(X)}[A(X)X]$, which has operation count u. After $B_1(X)$ is computed, $\mathrm{R}_{F_h(X)}[B_1(X)X]$ also has operation count u. There are also two binary additions (i.e., two XOR operations) in (22). Thus, the operation count required for computing $B(X)$ in (22) for the fast CRC is $2u + 2$.

Let us now determine the total operation counts t_b and t_f for computing the check tuple $P(X)$ for the basic CRC and the fast CRC, respectively. The CRC algorithm for computing $P(X)$, which is shown in Fig. 6, has a loop of n iterations. In addition to the operation count for $B(X)$, there is also one addition as indicated in line 4 of Fig. 6. Let l_n be the operation count for the loop overhead shown at line 2 of Fig. 6. We then have

$$t_b = n[l_n + 1 + s(u + l_s)], \tag{23}$$

$$t_f = n(l_n + 3 + 2u). \tag{24}$$

The basic CRC and the fast CRC require t_b and t_f operations, respectively, to compute the check tuple $P(X)$ for the input message that has ns bits, i.e., $t_b/(ns)$ and $t_f/(ns)$ operations are required per input bit. Recall that e_b and e_f denote the operation counts per input 8-bit byte required for computing the check tuple $P(X)$, for the basic CRC and the fast CRC, respectively. We then have $e_b = 8t_b/(ns)$ and $e_f = 8t_f/(ns)$. Using (23) and (24), we have

$$e_b = \frac{8t_b}{ns} = \frac{8[l_n + 1 + s(u + l_s)]}{s}, \tag{25}$$

$$e_f = \frac{8t_f}{ns} = \frac{8(l_n + 3 + 2u)}{s}, \tag{26}$$

$$\frac{e_b}{e_f} = \frac{t_b}{t_f} = \frac{l_n + 1 + s(u + l_s)}{l_n + 3 + 2u}. \tag{27}$$

Simple estimates are $t_b \approx nsu$ [by ignoring $l_n + 1$ and l_s in (23)] and $t_f \approx n2u$ [by ignoring $l_n + 3$ in (24)]. Substituting these into (27), we have

$$\frac{e_b}{e_f} = \frac{t_b}{t_f} \approx \frac{s}{2} = \frac{h}{2}, \tag{28}$$

i.e., the fast CRC is approximately $h/2$ times faster than the basic CRC.

4.2 CRC Complexity Under C Implementation

Figs. 7 and 8 show the C programs for the basic CRC and the fast CRC, respectively, which are based on Fig. 6 ($s = h$). For illustration, we let $s = 16$ in the figures, and $M(X) = X^{16} + X^{15} + X^2 + 1$ (which generates the CRC-16) in Fig. 7. However, the following results are also valid for other values of s and other generator polynomials.

We use the following two rules to count the number of software operations [19, Appendix A]: (R1) The *operation count* of a program statement is defined as the number of operations, other than the equal sign (=), that appear in that statement. (R2) For an if-statement, we average the operation count of the if-statement and the operation count of its alternative (e.g., an else-statement).

The nonzero operation count for each C program statement is recorded between the comment quotes ($/**/$). The programs show that $l_n = l_s = 2$. Using (20) of Remark 3, we have $u = 3$ if $\mathrm{msb}(A) = 0$ and $u = 4$ if $\mathrm{msb}(A) = 1$. Using rule (R2), we have $u = 3.5$ (which is the average of 3 and 4), as recorded in Figs. 7 and 8.

```
unsigned short basic_CRC (int n, unsigned short *Q)
{
int            i, j, s;
unsigned  short K, M, P;
s = 16;
M = 0x8005;     /* M = X^16+X^15+X^2+1 */
K = 0x8000;     /* K = 2^(h-1), h=s=16 */

P = 0;
for (i=0; i<n; i=i+1)                       /* 2 */
    {
    P = P ^ Q[i];                           /* 1 */

    for (j=0; j<s; j=j+1)                   /* 2 */
       {
       if ( (P&K) != 0 ) P = (P<<1) ^ M;  /* 3.5 */
       else              P =  P<<1;
       }
     }

return P;
}
```

Fig. 7. C program for the basic CRC ($s = h$).

Substituting these values of l_n, l_s, and u into (25) and (26), we obtain $e_b = 8(3 + 5.5s)/s$ and $e_f = 96/s$. Thus, we have

$$\frac{e_b}{e_f} = \frac{8(3 + 5.5h)}{96} = 0.25 + 0.458h, \tag{29}$$

which is within 10 percent of (28). For example, let $s = h = 16$. Then, $e_b = 8(3 + 5.5 \times 16)/16 = 45.5$ and $e_f = 96/16 = 6$. Thus, $e_b/e_f = 45.5/6 = 7.58$, i.e., the fast CRC is 7.58 times "faster" than the basic CRC. Further, if $s = h = 64$, then $e_f = 1.50, e_b = 44.4$, and $e_b/e_f = 29.6$, i.e., the fast 64-bit CRC is 29.6 times faster than the basic 64-bit CRC. These results are recorded in Fig. 9.

We now briefly present the complexity results for $s, h \in \{8, 16, 32, 64\}$, but without the restriction $s = h$. From (37) and (39) of [19, Appendix A], we have

$$e_b = \begin{cases} 8(4 + 5.5s)/s & \text{if } s < h, \\ 8(3 + 5.5s)/s & \text{if } s \geq h, \end{cases} \tag{30}$$

```
unsigned short fast_CRC (int n, unsigned short *Q)
{
int            i;
unsigned short A, C, F, K, P;
F = 0x7;        /* F = X^16+X^2+X+1 */
K = 0x8000;     /* K = 2^(h-1), h=s=16 */

P = 0;
for (i=0; i<n; i=i+1)                      /* 2 */
    {
    A = P ^ Q[i];                          /* 1 */

    if ( (A&K) != 0 ) C = (A<<1) ^ F;  /* 3.5 */
    else              C =  A<<1;

    if ( (C&K) != 0 ) P = (C<<1) ^ F;  /* 3.5 */
    else              P =  C<<1;

    P = P ^ C ^ A;                         /* 2 */
    }

return P;
}
```

Fig. 8. C program for the fast CRC ($s = h$).

	$s = 8$	$s = 16$	$s = 32$	$s = 64$
	e_b	e_b	e_b	e_b
	e_f	e_f	e_f	e_f
	e_b/e_f	e_b/e_f	e_b/e_f	e_b/e_f
$h = 8$	47.0	45.5	44.8	44.4
	12.0	28.0	36.0	40.0
	3.92	1.62	1.24	1.11
$h = 16$	48.0	45.5	44.8	44.4
	10.0	**6.00**	25.0	34.5
	4.80	7.58	1.79	1.29
$h = 32$	48.0	46.0	44.8	44.4
	10.0	5.00	**3.00**	23.5
	4.80	9.20	14.9	1.89
$h = 64$	48.0	46.0	45.0	44.4
	10.0	5.00	2.50	**1.50**
	4.80	9.20	18.0	29.6

Fig. 9. Software complexity for the basic CRCs (e_b) and the fast CRCs (e_f).

$$e_f = \begin{cases} 80/s & \text{if } s < h - 1, \\ 100/s & \text{if } s = h - 1, \\ 96/s & \text{if } s = h, \\ 8[12 + 5.5(s - h)]/s & \text{if } s > h. \end{cases} \tag{31}$$

As an example, consider a basic 16-bit CRC and the fast 16-bit CRC, which are used to protect an input message consisting of 8-bit bytes, i.e., $h = 16$ and $s = 8$. From the above formulas, we have $e_b = 8(4 + 5.5 \times 8)/8 = 48$ and $e_f = 80/8 = 10$. That is, the basic CRC and the fast CRC use 48 and 10 operations per input byte, respectively, to compute their check tuples. Thus, we have $e_b/e_f = 48/10 = 4.8$, i.e., the fast CRC is 4.8 times faster than the basic CRC. The values of e_b, e_f, and e_b/e_f for various (h, s) pairs are recorded in Fig. 9. The results show that the complexity of the basic CRCs is rather insensitive to the values of h and s, namely e_b varies from 44.4 to 48 (the variation is only 8.1 percent). In contrast, the complexity of the fast CRCs is very sensitive to the values of h and s, namely e_f varies from 1.50 up to 40.0.

For a given h, recall from Section 2.1 that we are free to choose the value of s. The complexity of the basic CRCs is rather insensitive to the choice of s. As seen in Fig. 9, when $h \in \{8, 16, 32, 64\}$, the complexity of the fast CRCs is fairly low when $s < h$, and is minimized when $s = h$. When $h \notin \{8, 16, 32, 64\}$, it is shown in [19, Appendix A] that the complexity of the fast CRCs is minimized (i.e., e_f is minimized) either at $s = h$ or at $s = h - 2$.

To summarize, we introduce the new family of CRC generator polynomials that have the explicit form $F_h(X) = X^h + X^2 + X + 1$, for all $h \geq 4$, as well as the new technique (15) for their implementation. This family includes $F_8(X)$, which generates the ATM CRC-8. For this particular CRC, by choosing $s = h = 8$, our new technique provides a new bitwise implementation that is 3.92 times faster than the basic bitwise technique (see Fig. 9).

Remark 4. There exist well-known techniques for reducing the operation counts used in CRC implementation. An example is the use of table lookup (at the cost of increased memory and cache usage), which is presented in [19, Appendix A]. Note that, to keep our C programs compact, readable, and general, we ignore software optimization techniques (such as loop unrolling) in our

C programs. However, these techniques certainly can be used to reduce the operation counts in the programs. For example, if loop unrolling is used (at the cost of code size expansion) in the inner for-loop of the C program in Fig. 7, then the index increment and the end-of-loop test are eliminated, i.e., the loop overhead l_s is reduced from $l_s = 2$ to $l_s = 0$. With loop unrolling (i.e., $l_s = 0$), it can be shown that (30) and (31) reduce to

$$e_b = \begin{cases} 8(4 + 3.5s)/s & \text{if } s < h, \\ 8(3 + 3.5s)/s & \text{if } s \geq h, \end{cases}$$

$$e_f = \begin{cases} 80/s & \text{if } s < h - 1, \\ 100/s & \text{if } s = h - 1, \\ 96/s & \text{if } s = h, \\ 8[12 + 3.5(s - h)]/s & \text{if } s > h. \end{cases}$$

4.3 Other Techniques for Error-Detection Codes

The complexity results for the basic CRC algorithm, which are rather insensitive to the input parameters s, h, and the form of the generator polynomial $M(X)$, are shown in Fig. 9. In particular, when $h = 16$ and $s = 8$, we have $e_b = 48$ operations per input byte. Our CRC software implementation in C for this case is shown [19, Appendix A, Fig. 11], which is more efficient than the one given in [2, pp. 555-556], which has 63 operations per input byte according to rules (R1) and (R2).

There are other CRC algorithms that are much faster than the basic algorithm. As expected, those algorithms are effective for some particular generator polynomials. For example, the clever "add and shift" algorithm of [4] is fast for the CRCs generated by $M_1(X) = X^{32} + X^{31} + X^8 + 1$ (for $h = 32$) and $M_2(X) = X^{64} + X^{63} + X^2 + 1$ (for $h = 64$), which are found by computer search [4]. According to rules (R1) and (R2) for determining the operation counts, these CRCs use 20 operations to process each tuple of $s = 32$ input bits (see [4, Fig. 2]). Thus, these CRCs use five operations per input byte. In contrast, from Fig. 9, for $s = 32$, our fast CRCs use only 3 and 2.5 operations per input byte for $h = 32$ and $h = 64$, respectively. Thus, our fast CRCs are faster than the above shift-and-add CRCs. Further, our fast 64-bit CRC is even much faster when $s = 64$, because it uses only 1.5 operations per input byte (see Fig. 9).

As mentioned in Section 1, alternatives to CRCs are checksums. Although checksums are weaker than CRCs, they can be substantially faster than CRCs. For example, let $s = h$ and consider the block-parity checksum. The check tuple $P(X)$ of this checksum is simply the sum of all the input tuples, i.e., $P(X) = \sum_{i=0}^{n-1} Q_i(X)$. As shown in [19, Section B.1], the operation count per input byte required for computing $P(X)$ of the checksum is $e = 24/s$. From (31), the fast CRC has $e_f = 96/s$. Thus, $e_f/e = 96/24 = 4$, i.e., the checksum is four times faster than the fast CRC.

5 Summary and Extension

Error control coding is essential for reliable transmission and storage, and CRCs are known to be effective for error detection. In software, an h-bit CRC is typically implemented by dividing the input message into s-tuples (i.e., blocks of s bits). The output CRC check bits are obtained by recursively carrying the polynomial division on these tuples.

Thus, the crucial part in CRC computation is the polynomial division on s-tuples. For the basic CRCs, this division requires s iterations, which may be expensive for many applications. A common technique for reducing the many steps during CRC computation is to use additional memory in the form of table lookup. In this paper, we introduce the fast h-bit CRCs, which are generated by $F_h(X) = X^h + X^2 + X + 1$, as well as the new technique (15) to implement them. Using our fast CRCs, the polynomial division on s-tuples requires only $\max(0, s - h + 2)$ iterations, which are much less than the s iterations required for the basic CRCs, as long as s is chosen such that $s - h + 2$ is much smaller than s. We study the computational complexity of the CRCs, which refers to the operation count per input byte required for computing the CRC check tuples. Our fast CRCs have low complexity and require no table lookup. For the important case $s = h$, the fast h-bit CRCs are approximately $h/2$ times faster than the basic h-bit CRCs.

As an illustration, we implement the CRCs in C programming language, and then study their computational complexity for the bitwise technique (i.e., without table lookup). We show that the complexity of the fast h-bit CRCs varies greatly with s, and is minimized either at $s = h - 2$ or at $s = h$. In contrast, the complexity of the basic h-bit CRCs varies little with s. Because modern computers typically process information in bytes or words, we also present the complexity results when s is restricted to multiples of byte size and word size.

In [19], we provide several extensions to the baseline ideas presented in this paper. In particular, we present the results for CRC table-lookup techniques, which illustrate tradeoffs between computational complexity and memory requirement. We show that when $s = h$, the fast CRCs can be made 20 percent faster by using tables of only four entries. We apply our new technique to some weaker CRCs to yield even faster CRCs, i.e., there are tradeoffs between speed and capability. Further, we use the new technique to construct some fast extended Hamming perfect codes. In particular, we construct h-bit non-CRC codes that not only have low complexity but also have the following optimal properties. They have the minimum distance $d = 4$, the burst-error-detecting capability $b = h$, and the maximum code length 2^{h-1}. We also apply the new technique to arbitrary CRCs, and then determine the conditions under which the new technique remains effective. In particular, the new technique is substantially faster than the basic technique for the CRC-64-ISO generated by $X^{64} + X^4 + X^3 + X + 1$. Finally, we show how the CRCs algorithms, which are originally designed for sequential implementation on a single processor, can be adapted for parallel implementation on multiple processors.

Acknowledgments

This work was supported in part by the US Office of Naval Research.

REFERENCES

[1] E.R. Berlekamp, *Algebraic Coding Theory.* McGraw-Hill, 1968.

[2] A. Binstock and J. Rex, *Practical Algorithms for Programmers.* Addison-Wesley, 1995.

[3] P. Farkas, "Comments on 'Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes'," *IEEE/ACM Trans. Networking,* vol. 3, no. 2, pp. 222-223, Apr. 1995.

[4] D.C. Feldmeier, "Fast Software Implementation of Error Detection Codes," *IEEE/ACM Trans. Networking,* vol. 3, no. 6, pp. 640-651, Dec. 1995.

[5] J.G. Fletcher, "An Arithmetic Checksum for Serial Transmissions," *IEEE Trans. Comm.,* vol. 30, no. 1, pp. 247-252, Jan. 1982.

[6] H.M. Ji and E. Killian, "Fast Parallel CRC Algorithm and Implementation on a Configurable Processor," *Proc. IEEE Int'l Conf. Comm. (ICC '02),* pp. 1813-1817, Apr./May 2002.

[7] T. Klove and V. Korzhik, *Error Detecting Codes: General Theory and Their Application in Feedback Communication Systems.* Kluwer Academic, 1995.

[8] D.E. Knuth, *The Art of Computer Programming,* vol. 2, third ed. Addison-Wesley, 1998.

[9] M.E. Kounavis and F.L. Berry, "A Systematic Approach to Building High Performance Software-Based CRC Generators," *Proc. 10th IEEE Symp. Computers and Comm. (ISCC '05),* pp. 855-862, June 2005.

[10] S. Lin and D.J. Costello Jr., *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, 1983.

[11] F.J. MacWilliams and N.J.A. Sloan, *The Theory of Error-Correcting Codes.* North-Holland, 1977.

[12] A.J. McAuley, "Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes," *IEEE/ACM Trans. Networking,* vol. 2, no. 1, pp. 16-22, Feb. 1994.

[13] G.D. Nguyen, "Error-Detection Codes: Algorithms and Fast Implementation," *IEEE Trans. Computers,* vol. 54, no. 1, pp. 1-11, Jan. 2005.

[14] A. Perez, "Byte-Wise CRC Calculations," *IEEE Micro,* vol. 3, no. 3, pp. 40-50, June 1983.

[15] T.V. Ramabadran and S.S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro,* vol. 8, no. 4, pp. 62-75, Aug. 1988.

[16] D.V. Sarwate, "Computation of Cyclic Redundancy Checks via Table-Lookup," *Comm. ACM,* vol. 31, no. 8, pp. 1008-1013, Aug. 1988.

[17] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of Checksums and CRC's over Real Data," *IEEE/ACM Trans. Networking,* vol. 6, no. 5, pp. 529-543, Oct. 1998.

[18] N. Zierler, "On $x^n + x + 1$ over GF(2)," *Information and Control,* vol. 16, no. 5, pp. 502-505, 1970.

[19] G.D. Nguyen, "Fast CRCs," *Supplemental Material,* http://doi.ieeecomputersociety.org/10.1109/TC.2009.83.

Gam D. Nguyen received the PhD degree in electrical engineering from the University of Maryland, College Park, in 1990. He is currently at the Naval Research Laboratory, Washington, District of Columbia. His research interests include communication systems and networks.

Supplemental Material for
Gam D. Nguyen, "Fast CRCs," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1321-1331, Oct. 2009
http://doi.ieeecomputersociety.org/10.1109/TC.2009.83

APPENDIX A CRC SOFTWARE IMPLEMENTATION AND COMPLEXITY EVALUATION

The purpose of this appendix is to present software implementation for the CRC algorithms as well as to evaluate their computational complexity. *Software* complexity of an algorithm refers to the number of operations (i.e., operation count) used to implement the algorithm. Consider an h-bit CRC, which is generated by a polynomial $M(X)$ of degree h. Our goal is to compute the check h-tuple $P(X)$ for an input message that consists of n tuples $Q_0(X), Q_1(X), \ldots, Q_{n-1}(X)$. Each tuple $Q_i(X)$ has s bits.

The CRC can be implemented by any of the 4 algorithms shown in Figs. 1-4. Although the value of h is fixed, we are free to choose the value of s. Algorithms 1 and 3 are for $s < h$, whereas Algorithms 2 and 4 are for $s \geq h$. One algorithm can be faster than another, depending the value of s and the form of $M(X)$. For example, Remark 2 shows that, for bitwise implementation, Algorithm 4 is faster than Algorithm 2 when $s \geq h$. Thus, we will use Algorithm 4 for bitwise implementation when $s \geq h$, as indicated in Fig. 10. As stated in Theorem 1, Algorithm 3 must be used for the fast CRCs when $s < h$. Fig. 10 lists the CRC algorithms that are used in our software implementation.

We recognize that accurate software evaluation is complicated, and requires experiments with different processors, memory organizations, programming languages, and compilers. Other complicating factors include programming styles and the extend the CRCs must share with (or compete against) other concurrent/interupting programs.

Instead of dealing with the complex issues mentioned above, which are beyond the scope of this paper, we simply use *software operation counts* for our complexity evaluation. Our technique of software comparison is as follows. We write a program (e.g., in C) for each CRC. We then use the operation count as the primary measure of complexity, and a CRC is said to be "faster" than another if it has lower operation count.

We now determine the software complexity of the CRC algorithms, which refers to the operation count per input message byte required for computing the check h-tuple. Let us examine Algorithms 1-4 (shown in Figs. 1-4). For each algorithm, the check tuple $P(X)$ is computed by using a loop that computes $B(X)$ for n times, where $B(X)$ is given in Definition 1 for the basic CRC and in Theorem 1 for the fast CRC. In addition to $B(X)$, we also need to compute all the other terms *inside* the loop (which include the loop overhead). Let r and x be the operation counts required for computing $B(X)$ and the other terms *inside* the loop, respectively. Let y be the operation count required for computing the terms *outside* the loop. Further, for each algorithm, let t be the total operation count required for computing the CRC check tuple from the input message that consists of n tuples. We then have $t = (x + r)n + y$.

Let e be the operation count per input *byte* required for computing the check h-tuple. Each byte has 8 bits. Because t is the operation count for computing the check h-tuple from the ns input message bits, we have $e = 8t/(ns) = 8[(x + r)n + y]/(ns)$, i.e.,

$$e = \frac{8(x + r)}{s} + \frac{8y}{ns} \tag{32}$$

In the following, we consider h, s, and n to be independent variables, and our goal is to compute e in terms of h, s, and n for both the basic CRCs and the fast CRCs. That is, we can write $e = e(s, h, n)$. To compute e, we need to determine r, x, and y, to which we add the subscripts b and f when they refer to the basic CRCs and the fast CRCs, respectively. That is, r_b, x_b, y_b, and e_b refer to the basic CRCs, while r_f, x_f, y_f, and e_f refer to the fast CRCs.

We present CRC implementation with and without table lookup. Our software programs are for w-bit computers that satisfy $s \leq w$ and $h \leq w$ (however, we allow the possibility that $h + s > w$). For example, 32-bit computers are for $s, h \leq 32$ bits, while 64-bit computers are for $s, h \leq 64$ bits (future 128-bit computers are for $s, h \leq 128$ bits). To be specific, we implement the CRC algorithms in C, which is a highly portable general-purpose computer programming language (certainly, they can also be implemented in other computer languages). We use the following 2 simple rules to count the number of software operations [13]:

(R1) The *operation count* of a program statement is defined as the number of operations, other than the equal sign (=), that appear in that statement.

(R2) For an if-statement, we average the operation count of the if-statement and the operation count of its alternative (e.g., an else-statement).

Let us consider examples on how to use rule (R1). The statement `C = (A<<1)∧F` will count as 2 operations (<< and ∧). Note that "=" does not count as an operation. Next, consider the statement `for(i=0; i<n; i=i+1){ }`. This implements a (null) loop of n iterations, each iteration has 2 operations (< and +). Thus, the total operation count for this loop statement is $2n$. The for-loop above is equivalent to the while-loop `i=0; while(i<n){i=i+1;}` which, of course, also has $2n$ operations.

We now show examples about rule (R2). Suppose that `K = 1`, and consider the following 2 statements:

```
if ((A&K) != 0) C =(A<<1)∧F;
else            C = A<<1;
```

Here, the if-statement has 4 operations (&, !=, <<, ∧), and the else-statement has 3 operations (&, !=, <<). Thus, the above 2 statements can be considered as a single statement that has 3.5 operations (i.e., the average of 4 and 3).

The above 2 statements are equivalent to the the following 2 statements:

```
C = A<<1;
if ((A&K) != 0) C = C∧F;
```

Here, the first statement has 1 operation, and the second statement has 2.5 operations. Thus, the 2 statements together also have 3.5 operations as expected. Note that `(A&K)`$\in \{0, 1\}$, because `K = 1`. Here, for simplicity, we assume that `(A&K)` takes the values 0 and 1 with equal probability of 1/2. Suppose now that `K = 3`. We then have`(A&K)`$\in \{0, 1, 2, 3\}$. By assuming that `(A&K)` takes the values 0, 1, 2, and 3 with equal probability of 1/4, the above if-statement (which has 4 operations) is executed with probability 3/4 and the else-statement (which has 3 operations) is executed with probability 1/4. Thus, these if-else statements can be considered as a single statement that has $4 \times 3/4 + 3 \times 1/4 = 3.75$ operations.

	bitwise	table lookup
basic CRC	Algo. 1 ($s < h$) Algo. 4 ($s \geq h$)	Algo. 3 ($s < h$) Algo. 2 ($s \geq h$)
fast CRC	Algo. 3 ($s < h$) Algo. 4 ($s \geq h$)	Algo. 3 ($s < h$) Algo. 2 ($s \geq h$)

Fig. 10 CRC algorithms used in software implementation

Remark 5. Rules (R1) and (R2) serve as a simple technique for comparing the complexity of different CRCs, i.e., they will be used to obtain a first-order estimation of the ratio e_b/e_f. These rules are intended only for CRC algorithms that are implemented in C, and not for other types of algorithms or other programming languages. As seen in the following, our CRC software implementation uses only a small number of elementary C operators (namely, +, <<, >>, =, ==, !=, <, <=, &, and ∧) and C keywords (namely, *char, short, int, long, unsigned, if, else, for, while,* and *return*). Our following C programs for the CRCs are written in a style that is intended to be simple and straightforward. See also Remark 6.

Other techniques for counting operations are also possible. For example, consider rule (R1′), which is defined as rule (R1), but also counts the equal sign (=) as an operation. Let e'_b and e'_f denote the resulting operation counts under (R1′). We must have $e'_b > e_b$ and $e'_f > e_f$. Although the difference between e_b and e'_b (as well as between e_f and e'_f) can be significant, the difference between the ratios e_b/e_f and e'_b/e'_f are typically not significant. For example, let $s = h = 32$. From Fig. 9, we have $e_b = 44.8$, $e_f = 3$, and $e_b/e_f = 14.9$. Under rule (R1′), it can be shown that $e'_b = 61.5$, $e'_f = 4.25$, and $e'_b/e'_f = 14.5$, i.e., $e'_b/e'_f \approx e_b/e_f$. Note that rule (R1), which is used in this paper, is slightly simpler to use than rule (R1′). Thus, our technique for counting software operations is reasonable for the purpose of *complexity comparison*, i.e., we are more interested in the ratio e_b/e_f, rather than in e_b and e_f.

Here, for simplicity, we assign the same unit cost to each operation. A more elaborate technique would assign different costs to different operations. However, this assignment depends on many factors (such as computer hardware, operating system, processor architecture, and memory organization), which are outside the focus of this paper. □

Let us now compute x and y in (32). The computation of r is deferred to later subsections. First, consider Fig. 11, which shows the C program for bitwise implementation of the basic CRC for the case $s < h$. As indicated in Fig. 10, this program is based on Algorithm 1. In this program, we assume that $h \in \{8, 16, 32, 64\}$, i.e., h is the size (in bits) of one of the natural *unsigned types* of C: unsigned char, unsigned short int, unsigned int, or unsigned long int. The input is the n message s-tuples $Q[0], Q[1], \ldots, Q[n-1]$, and the output is the CRC check h-tuple P.

We then apply rules (R1) and (R2) to the program shown in Fig. 11 to obtain the desired operation counts. The non-zero operation count for each program statement is recorded between the comment quotes

(/* */). Recall that the total operation count for computing the check tuple P from the n input tuples is $t_b = (x_b + r_b)n + y_b$. Here, r_b is the operation count required for computing $B(X)$, which is inside the loop indexed by i, x_b is the operation count required for computing all the other terms in the loop besides $B(X)$, and y_b is the operation count required for computing all the terms outside the loop. From Fig. 11, we have $x_b = 4$ and $y_b = 0$.

To summarize, for $h \in \{8, 16, 32, 64\}$ and $s < h$, we have $x_b = 4$ and $y_b = 0$, which are recorded in Fig. 13. For illustration, we let $h = 16$, $s = 8$ and $M(X) = X^{16} + X^{15} + X^2 + 1$ (which generates the well-known CRC-16) in Fig. 11. When $h \notin \{8, 16, 32, 64\}$ and $s < h$, the computational complexity is slightly higher, namely, $x_b = 4$ and $y_b = 1$.

Fig. 7 shows the C program for the basic CRC when $s = h$. Next, consider Fig. 12, which shows the C program for the fast CRC when $h \in \{8, 16, 32, 64\}$ and $s < h - 1$. It can be shown that $x_f = 6$ and $y_f = 0$ for this case. Again, for illustration, we let $h = 16$ and $s = 8$ in Fig 12. Similarly, we can compute the values of x and y for all the cases for both the basic CRCs and the fast CRCs. The results are summarized in Fig. 13.

Using Fig. 13, the expression (32) can be simplified as follows. Let z be the ratio of the 2 terms on the right-hand side of (32), i.e.,

$$\begin{aligned} z &= \frac{8(x+r)/s}{(8y)/(ns)} \\ &= \frac{(x+r)n}{y} \end{aligned}$$

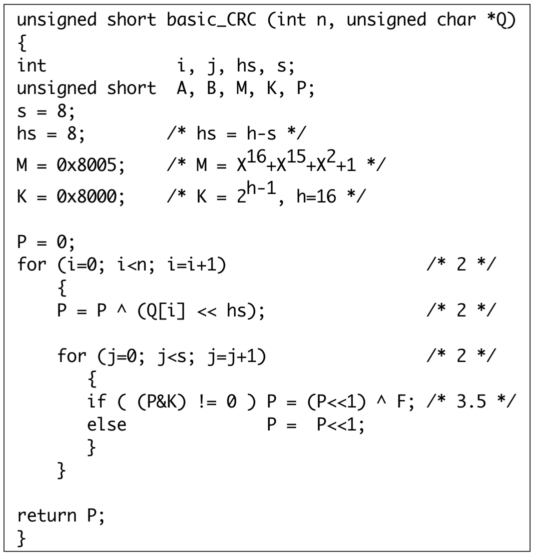
From Fig. 13, we have $0 \leq y \leq 1$ and $x \geq 3$, which implies that $z \geq (x + r)n \geq (3 + r)n \geq 3n$. Thus, $8y/(ns)$ is much smaller than $8(x + r)/s$, because we assume in this paper that n is not too small (i.e., we assume that $n > 4$). Thus, the term $8y/(ns)$ can be dropped from (32). The operation count per input byte required for computing the CRC check h-tuple then simplifies to

$$e = \frac{8(x+r)}{s} \tag{33}$$

where x is determined from Fig. 13, which depends only on s and h, i.e., $x = x(s, h)$. Recall that r denotes the operation count required for computing $B(X)$, which also depends only on s and h [see (11)], i.e., $r = r(s, h)$. It follows from (33) that e now also depends only on s and h, i.e., $e = e(s, h)$. From (33), we also have

$$\frac{e_b}{e_f} = \frac{x_b + r_b}{x_f + r_f}$$

where x_b and x_f are given in Fig. 13. In the following, using rules (R1) and (R2), we compute r_b and r_f for both the bitwise and the table-lookup techniques.



```
unsigned short basic_CRC (int n, unsigned char *Q)
{
int             i, j, hs, s;
unsigned short  A, B, M, K, P;
s = 8;
hs = 8;         /* hs = h-s */
M = 0x8005;     /* M = X^16+X^15+X^2+1 */
K = 0x8000;     /* K = 2^h-1, h=16 */

P = 0;
for (i=0; i<n; i=i+1)                    /* 2 */
    {
    P = P ^ (Q[i] << hs);                /* 2 */

    for (j=0; j<s; j=j+1)                /* 2 */
       {
       if ( (P&K) != 0 ) P = (P<<1) ^ F; /* 3.5 */
       else              P =  P<<1;
       }
    }

return P;
}
```

Fig. 11 C program for the basic h-bit CRC ($s < h$)

```
unsigned short fast_CRC (int n, unsigned char *Q)
{
int             i, hs, s;
unsigned short  A, B, P;
s = 8;
hs = 8;         /* hs = h-s, h = 16 */

P = 0;
for (i=0; i<n; i=i+1)         /* 2 */
    {
    A = (P>>hs) ^ Q[i];       /* 2 */
    B = (A<<2) ^ (A<<1) ^ A;  /* 4 */
    P = B ^ (P<<s);           /* 2 */
    }

return P;
}
```

Fig. 12 C program for the fast h-bit CRC ($s < h - 1$)

	x	y
Algo. 1	4	0 if $h = 8, 16, 32, 64$
($s < h$)		1 if $h \neq 8, 16, 32, 64$
Algo. 2	3 if $s = h$	0
($s \geq h$)	4 if $s > h$	0
Algo. 3	6 if $h = 8, 16, 32, 64$	0 if $h = 8, 16, 32, 64$
($s < h$)	7 if $h \neq 8, 16, 32, 64$	1 if $h \neq 8, 16, 32, 64$
Algo. 4	3	0 if $s = h$
($s \geq h$)		1 if $s > h$

Fig. 13 Values of x and y

A.1 CRC Software Implementation: Bitwise Technique (Without Table Lookup)

According to Fig. 10, the bitwise implementation of the the basic CRCs uses Algorithm 1 for $s < h$, and Algorithm 4 for $s \geq h$. From Fig. 13, we then have

$$x_b = \begin{cases} 4 & \text{if } s < h \\ 3 & \text{if } s \geq h \end{cases}$$

Substituting x_b into (33), we have

$$e_b = \begin{cases} 8(4 + r_b)/s & \text{if } s < h \\ 8(3 + r_b)/s & \text{if } s \geq h \end{cases} \tag{34}$$

where r_b denotes the operation count required for computing $B(X)$ of the basic CRCs.

According to Fig. 10, the bitwise implementation of the the fast CRCs uses Algorithm 3 for $s < h$, and Algorithm 4 for $s \geq h$. From Fig. 13, we then have

$$x_f = \begin{cases} 6 & \text{if } s < h \text{ and } h = 8, 16, 32, 64 \\ 7 & \text{if } s < h \text{ and } h \neq 8, 16, 32, 64 \\ 3 & \text{if } s \geq h \end{cases}$$

Substituting x_f into (33), we have

$$e_f = \begin{cases} 8(6 + r_f)/s & \text{if } s < h \text{ and } h = 8, 16, 32, 64 \\ 8(7 + r_f)/s & \text{if } s < h \text{ and } h \neq 8, 16, 32, 64 \\ 8(3 + r_f)/s & \text{if } s \geq h \end{cases} \tag{35}$$

where r_f denotes the operation count required for computing $B(X)$ of the fast CRCs. Both r_b and r_f are computed in the following subsections.

A.1.1 Basic CRCs

Recall that $B(X)$ for the basic CRCs is given in Definition 1. First, consider the case $s < h$, and let us revisit Fig. 11. This figure contains the loop (indexed by j) for computing $B(X)$, which is based on Remark 3. The figure shows that the operation count required for computing $B(X)$ is $r_b = 5.5s$. Next, for the case $s \geq h$, it can also be shown that $r_b = 5.5s$ (see Fig. 7). To summarize, we have

$$r_b = 5.5s \tag{36}$$

Substitute (36) into (34) we have

$$e_b = \begin{cases} 8(4 + 5.5s)/s & \text{if } s < h \\ 8(3 + 5.5s)/s & \text{if } s \geq h \end{cases} \tag{37}$$

Note that (36) is derived from the C programs that do not use loop unrolling (which is also the case for the C programs presented in [2]). If loop unrolling is used, (36) reduces to $r_b = 3.5s$.

Here, our software implementations of the basic CRCs are general, i.e., they are applicable to all generator polynomials $M(X)$ and to a wide range of processor architectures. For some specific generator polynomials that have some desirable properties, alternative implementations (such as shift and add [4], and on the fly [14, 15]) may have lower complexity. Thus, we concentrate on the general nature of the algorithms rather than attempting to deal with specific types of generator polynomials. Also, for our C programs, we are more concerned with their readability and less concerned with optimization techniques such as loop unrolling and use of register variables (see Remark 4).

A.1.2 Fast CRCs

Recall that $B(X)$ of the fast CRCs is given in Theorem 1. First, assume that $s < h - 1$. The C program for this case is shown in Fig. 12, which contains the procedure for computing $B(X)$. Applying rules (R1) and (R2) to Fig. 12, we observe that the operation count required for computing $B(X)$ is $r_f = 4$. Next, assume that $s = h$. The C program for this case is shown in Fig. 8, which yields $r_f = 9$. The C programs for all the other cases can also be written, and the resulting software complexity can also be determined. Following is the list of the operation counts for all the cases:

$$r_f = \begin{cases} 4 & \text{if } s < h - 1 \\ 6.5 & \text{if } s = h - 1 \\ 9 & \text{if } s = h \\ 9 + 5.5(s - h) & \text{if } s > h \end{cases} \tag{38}$$

Substituting (38) into (35), we have

$$e_f = \begin{cases} 80/s & \text{if } s < h - 1 \text{ and } h = 8, 16, 32, 64 \\ 88/s & \text{if } s < h - 1 \text{ and } h \neq 8, 16, 32, 64 \\ 100/s & \text{if } s = h - 1 \text{ and } h = 8, 16, 32, 64 \\ 108/s & \text{if } s = h - 1 \text{ and } h \neq 8, 16, 32, 64 \\ 96/s & \text{if } s = h \\ 8[12 + 5.5(s - h)]/s & \text{if } s > h \end{cases} \tag{39}$$

The operation count per input byte e_f for the fast h-bit CRC given in (39) is a function of s, which is the size of each input tuple $Q_i(X)$. We now determine the value of s that minimizes e_f. These optimal values are denoted by s^* and e_f^*.

First, assume that $h \in \{8, 16, 32, 64\}$. For each $h \in \{8, 16, 32, 64\}$, we can search for an $s \in \{1, 2, \ldots, 64\}$ such that e_f in (39) is minimized. Our search shows that

$$e_f^* = \begin{cases} 80/(h - 2) & \text{if } h = 16, 32, 64 \\ 96/h & \text{if } h = 8 \end{cases} \tag{40}$$

which is achieved when

$$s^* = \begin{cases} h - 2 & \text{if } h = 16, 32, 64 \\ h & \text{if } h = 8 \end{cases} \tag{41}$$

Next, assume that $h \notin \{8, 16, 32, 64\}$. For each $h \in \{8, 16, 32, 64\}$, $4 \leq h \leq 64$, we can search for an $s \in \{1, 2, \ldots, 64\}$ such that e_f in (39) is minimized. Our search shows that

$$e_f^* = \begin{cases} 88/(h-2) & \text{if } h > 24 \\ 96/h & \text{if } 4 \le h \le 24 \end{cases} \tag{42}$$

which is achieved when

$$s^* = \begin{cases} h-2 & \text{if } h > 24 \\ h & \text{if } 4 \le h \le 24 \end{cases} \tag{43}$$

Thus, (41) and (43) show that the complexity of the fast h-bit CRCs is minimized (i.e., e_f is minimized) at either $s = h$ or $s = h - 2$, where s is the size of each input tuple $Q_i(X)$.

For example, by letting $h = 16$, the optimal size for each input tuple $Q_i(X)$ is $s^* = h - 2 = 14$ [by (41)], and the corresponding minimum operation count is $e_f^* = 80/(h-2) = 80/14 = 5.71$ [by (40)]. Information on computers is typically organized in bytes or words. Thus, it is of interest to determine the optimal value of e_f when s is restricted to a multiple of byte size and word size, i.e., when s is a multiple of 8, 16, 32, 64. These optimal values, which are obtained from (39), are shown in Fig. 14.

Recall that $e_f = e_f(s, h)$, i.e., e_f is a function of s and h. In Fig. 14, for a given h, $s^{(\mathrm{opt})}$ denotes the value of s, $1 \le s \le 64$, that minimizes $e_f(s, h)$, and the corresponding minimum $e_f(s, h)$ is denoted by $e_f^{(\mathrm{opt})}$. Thus, we have $e_f^{(\mathrm{opt})} = e_f(s^{(\mathrm{opt})}, h) \le e_f(s, h)$ for all $1 \le s \le 64$. Similarly, $s^{(\mathrm{byte})}$ denotes the value of $s \in \{8, 16, 24, 32, 40, 48, 56, 64\}$ that minimizes $e_f(s, h)$, and the corresponding minimum $e_f(s, h)$ is denoted by $e_f^{(\mathrm{byte})}$. Finally, $s^{(\mathrm{word})}$ denotes the value of $s \in \{8, 16, 32, 64\}$ that minimizes $e_f(s, h)$, and the corresponding minimum $e_f(s, h)$ is denoted by $e_f^{(\mathrm{word})}$. For example, by letting $h = 64$, we have $s^{(\mathrm{opt})} = 62$, $e_f^{(\mathrm{opt})} = 1.29$, $s^{(\mathrm{byte})} = 56$, $e_f^{(\mathrm{byte})} = 1.43$, $s^{(\mathrm{word})} = 64$, $e_f^{(\mathrm{word})} = 1.50$. In general, we must have $e_f^{(\mathrm{opt})} \le e_f^{(\mathrm{byte})} \le e_f^{(\mathrm{word})}$.

h	$s^{(\mathrm{opt})}$ $e_f^{(\mathrm{opt})}$	$s^{(\mathrm{byte})}$ $e_f^{(\mathrm{byte})}$	$s^{(\mathrm{word})}$ $e_f^{(\mathrm{word})}$
4	4 24.0	8 34.0	8 34.0
6	6 16.0	8 23.0	8 23.0
8	8 12.0	8 12.0	8 12.0
10	10 9.60	8 11.0	8 11.0
12	12 8.00	8 11.0	8 11.0
16	14 5.71	16 6.00	16 6.00
20	20 4.80	16 5.50	16 5.50
24	22 4.00	24 4.00	16 5.50
32	30 2.67	32 3.00	32 3.00
40	38 2.32	40 2.40	32 2.75
48	46 1.91	48 2.00	32 2.75
56	54 1.63	56 1.71	32 2.75
64	62 1.29	56 1.43	64 1.50

Fig. 14 The optimal values of s and e_f for the h-bit fast CRCs, when $s \in \{1, 2, \ldots, 63, 64\}$, when $s \in \{8, 16, 24, 32, 40, 48, 56, 64\}$, and when $s \in \{8, 16, 32, 64\}$

Remark 6. Our C programs for the CRCs, which follow directly from the pseudocodes in Figs. 1-4, are written in a style that is intended to be simple and straightforward. For readability, we use an array (instead of a pointer) for the input s-tuples Q_i. We also avoid using any C syntax that obscures the operation counts. For example, the more explicit syntax `if((P&K)!=0)` is used instead of the shorthand `if(P&K)`. Although these 2 expressions are equivalent, the former shows 2 operations more clearly. If desired, these C programs can be rewritten in pointer and shorthand style, for example, as shown in Figs. 15 and 16, which are equivalent to Figs. 7 and 8, respectively. □

```
#define M  0x8005    /* M = X^16+X^15+X^2+1 */
#define K  0x8000    /* K = 2^h-1, h=s=16 */
#define s  16

unsigned short basic_CRC (int n, unsigned short *Q)
{
register int            j, s;
register unsigned short K, M, P, *Qi, *Qn;

Qi = Q;
Qn = Q + n;                          /* 1 */
P = 0;
while (Qi < Qn)                      /* 1 */
    {
    P ^= *Qi++;                      /* 2 */
    for (j=0; j<s; j++)              /* 2 */
       {
       if (P&K)  P = (P<<1) ^ M;     /* 3.5 */
       else      P =  P<<1;
       }
    }

return P;
}
```

Fig. 15 C program for the basic h-bit CRC $(s = h)$ in pointer style

```
#define F  0x7       /* F = X^16+X^2+X+1 */
#define K  0x8000    /* K = 2^h-1, h=s=16 */
#define s  16

unsigned short fast_CRC (int n, unsigned short *Q)
{
register unsigned short A, C, F, K, P, *Qi, *Qn;

Qi = Q;
Qn = Q + n;                      /* 1 */
P = 0;
while (Qi < Qn)                  /* 1 */
    {
    A = P ^ *Qi++;               /* 2 */

    if (A&K) C = (A<<1) ^ F;     /* 3.5 */
    else     C =  A<<1;

    if (C&K) P = (C<<1) ^ F;     /* 3.5 */
    else     P =  C<<1;

    P ^= C ^ A;                  /* 2 */
    }

return P;
}
```

Fig. 16 C program for the fast h-bit CRC $(s = h)$ in pointer style

Remark 7. When s is small, we can compute $B(X) = \mathrm{R}_{M(X)}\left[A(X)X^h\right]$, where degree$(A(X)) < s$, using a series of if-else statements as follows. For example, suppose that $s = 2$ and $M(X) = F_h(X) = X^h+X^2+X+1$. Then $A(X) \in \{0, 1, X, X+1\}$, and it can be shown that

$$B(X) = \begin{cases} 0 & \text{if } A(X) = 0 \\ X^2+X+1 & \text{if } A(X) = 1 \\ X^3+X^2+X & \text{if } A(X) = X \\ X^3+1 & \text{if } A(X) = X+1 \end{cases}$$

Note that polynomials can also be represented as integer numbers, e.g., the polynomial $X^3 + X^2 + X$ is equivalent to the decimal number 14. Thus, $B(X)$ can be computed using the C program segment shown in Fig. 17. Applying rules (R1) and (R2) to this C program segment, the operation count for computing $B(X)$ is 1, 2, 3, or 3 if $A(X)$ is 0, 1, 2, or 3 (in integer representation), respectively. We now assume that the bits 0 and 1 of the input message occur equally likely. Thus, $A(X)$ assumes one of the values 0, 1, 2, 3 with equal probability of 1/4. Then, on the average, the operation count for computing $B(X)$ is $(1+2+3+3)/4 = 2.25$. In general, this technique for computing $B(X)$ can be applied to any generator polynomial $M(X)$.

Let k denote the operation count required for computing $B(X) = \mathrm{R}_{M(X)}\left[A(X)X^h\right]$ using this if-else technique, where degree$(A(X)) < s$. Note that k depends on s, i.e., $k = k(s)$. As shown above, we then have $k(2) = 2.25$, which is smaller than both r_b and r_f given in (36) and (38). In general, it can be shown that $k(s) = 2^{s-1} + 2^{-1} - 2^{-s}$ for $s \geq 1$. In particular, $k(1) = 1$. Thus, this if-else technique is effective for small s, such as $s = 1, 2$, or 3. However, in this paper, we are mainly concerned with the case $s \geq 8$, which is more commonly used in practice. For this case, $k(s)$ is much greater than both r_b and r_f. Thus, when $s \geq 8$, the if-else technique is much more expensive than the basic and the new techniques, and it will not be discussed further in this paper. Note also that this if-else technique is different from the table-lookup technique (which will be discussed later). □

```
if (A == 0) B = 0;    /* 2.25 */
else
   {
   if (A == 1) B = 7;
   else
      {
      if (A == 2) B = 14;
      else        B = 9;
      }
   }
```

Fig. 17 C program segment for computing $B(X) = \mathrm{R}_{F_h(X)}\left[A(X)X^h\right]$ for $s = 2$

Remark 8. Consider an input message $U(X)$, which is protected by an h-bit CRC. Recall that we implement this CRC by first dividing the input message into n s-tuples $Q_i(X)$, i.e., we have $U(X) = (Q_0(X), Q_1(X), \ldots, Q_{n-1}(X))$. These s-tuples $Q_i(X)$ then become the input to one of the CRC algorithms. Fig. 9 shows that the complexity of the basic CRCs is rather insensitive to the values of s, whereas the complexity of the fast CRCs is very sensitive to the values of s. Recall that the operation count per input byte e_f in (39) is a function of s and h, i.e., $e_f = e_f(s,h)$. For example, Fig. 9 shows that $e_f(8,16) = 10$ and $e_f(16,16) = 6$, i.e., $e_f(16,16) < e_f(8,16)$.

So far, we do not address the cost of obtaining the tuples $Q_i(X)$. We now address the impact of this cost by considering the fast 16-bit CRC, i.e., $h = 16$. Suppose that the input message $U(X)$ originally consists of m bytes, $m \geq 4$, denoted by $I_0(X), I_1(X), \ldots, I_{m-1}(X)$. Each $I_i(X)$ is an 8-tuple. Thus, we need to organize the bytes $I_j(X)$ into the s-tuples $Q_i(X)$. One technique is to simply set $Q_i(X) = I_i(X)$, i.e., each $Q_i(X)$ is an 8-tuple. Let e be the operation count per input byte required for CRC encoding. We then have $s = 8$, and hence $e = e_f(8,16) = 10$.

An alternative technique is first to pair 2 adjacent input bytes to form 16-bit tuples from which the check bits are then computed. More precisely, we now let $s = 16$ and define the new 16-tuples $Q_i(X)$ by

$$Q_0(X) = \begin{cases} (I_0(X), I_1(X)) & \text{if } m \text{ is even} \\ (0, I_0(X)) & \text{if } m \text{ is odd} \end{cases}$$

and

$$Q_i(X) = \begin{cases} (I_{2i}(X), I_{2i+1}(X)) & \text{if } m \text{ is even} \\ (I_{2i-1}(X), I_{2i}(X)) & \text{if } m \text{ is odd} \end{cases}$$

for

$$0 < i \leq \begin{cases} (m-2)/2 & \text{if } m \text{ is even} \\ (m-1)/2 & \text{if } m \text{ is odd} \end{cases}$$

The algorithm for pairing the bytes and then computing the fast 16-bit CRC is shown in Fig. 18. Using this algorithm, it can be shown that the operation count per input byte is $e = 7.5$, which is lower than $e_f(8,16) = 10$ of the non-pairing technique. Note that $e = 7.5 > e_f(16,16) = 6$, because of the additional cost for pairing the input bytes to form the new 16-bit tuples to be used for the CRC computation. □

```
if ($m$ is even)
    $\{Q = I_0X^8 + I_1;\ i = 2;\}$
else
    $\{Q = I_0;\ i = 1;\}$
$B = \mathrm{R}_{F_{16}}\left[Q(X^2 + X + 1)\right];$
while ($i < m - 1$)
    {
    $Q = I_iX^8;\ i = i + 1;$
    $Q = Q + I_i;\ i = i + 1;$
    $A = B + Q;$
    $B = \mathrm{R}_{F_{16}}\left[A(X^2 + X + 1)\right];$
    }
$P = B;$
return $P$;
```

Fig. 18 Algorithm for computing the fast 16-bit CRC directly from the m input bytes I_i

A.2 CRC Software Implementation: Table-Lookup Technique

Recall that the complexity of the fast CRCs is low even without using table lookup. With table lookup, the operation count is reduced at the cost of additional memory resource. Although our focus in this paper is on bitwise algorithms, we now also present table-lookup algorithms to illustrate tradeoffs between operation count and table size. Our formulation and results here are straightforward generalizations or variations of well-known results, which are available in [2, 4, 9, 14, 15, 16]. Note that, with table lookup, speed directly correlates with operation count under ideal conditions (e.g., the table is stored in the fastest cache, and there is no cache miss). Otherwise, speed may not correlate directly with operation count (e.g., when the impact of cache miss is not negligible [4]).

For table-lookup implementation, according to Fig. 10, we use Algorithm 3 (when $s < h$) and Algorithm 2 (when $s \geq h$) for both the basic CRCs and the fast CRCs. From (11), we then have

$$B(X) = \mathrm{R}_{M(X)}\left[A(X)X^h\right]$$

where degree$(A(X)) < s$. In the following, $B(X)$ is computed by table lookup. Let g_b and g_f be the total number of table entries for the basic CRCs and the fast CRCs, respectively.

A.2.1 Basic CRCs

According to Fig. 10, Algorithms 2 and 3 are used for the basic CRCs. Substituting the values of x from Fig. 13 for Algorithms 2 and 3 into (33), we have

$$e_b = \begin{cases} 8(6 + r_b)/s & \text{if } s < h \text{ and } h = 8, 16, 32, 64 \\ 8(7 + r_b)/s & \text{if } s < h \text{ and } h \neq 8, 16, 32, 64 \\ 8(3 + r_b)/s & \text{if } s = h \\ 8(4 + r_b)/s & \text{if } s > h \end{cases} \tag{44}$$

where r_b is the operation count required for computing $B(X)$ via table lookup. The required tables are defined below. First, we write

$$s = t_1 + t_2 + \cdots + t_m$$

for some m and t_i such that $1 \leq m \leq s$ and $1 \leq t_i \leq s$ $(i = 1, 2, \ldots, m)$. Next, we decompose $A(X)$ into m polynomials $A_1(X), A_2(X), \ldots, A_{m-1}(X), A_m(X)$ such that

$$\begin{aligned} A(X) &= A_1(X)X^{(t_2+s_3+\cdots+t_m)} + A_2(X)X^{(t_3+s_4+\cdots+t_m)} + \cdots + A_{m-1}(X)X^{t_m} + A_m(X) \\ &= \sum_{i=1}^{m-1} A_i(X)X^{(t_{i+1}+\cdots+t_m)} + A_m(X) \end{aligned}$$

with degree$(A_i(X)) < t_i$, $i = 1, 2, \ldots, m$. We then have

$$\begin{aligned}
B(X) &= \mathrm{R}_{M(X)}\left[A(X)X^h\right] \\
&= \mathrm{R}_{M(X)}\left[\sum_{i=1}^{m-1} A_i(X)X^{(t_{i+1}+\cdots+t_m+h)} + A_m(X)X^h\right] \\
&= \sum_{i=1}^{m-1} \mathrm{R}_{M(X)}\left[A_i(X)X^{(t_{i+1}+\cdots+t_m+h)}\right] + \mathrm{R}_{M(X)}\left[A_m(X)X^h\right] \\
&= \sum_{i=1}^{m-1} T_i[A_i] + T_m[A_m]
\end{aligned} \tag{45}$$

where the tables $T_i[\ ]$ are defined by

$$T_i[A_i] = \begin{cases} \mathrm{R}_{M(X)}\left[A_i(X)X^{(t_{i+1}+\cdots+t_m+h)}\right] & \text{if } 1 \le i < m \\ \mathrm{R}_{M(X)}\left[A_m(X)X^h\right] & \text{if } i = m \end{cases} \tag{46}$$

where A_i denotes the t_i-tuple that is composed of the binary coefficients of $A_i(X)$. For example, if $t_i = 4$ and $A_i(X) = X^2 + 1$, then $A_i = (0101)$, which is equivalent to the decimal integer 5.

Thus, regardless of whether $s < h$ or $s \ge h$, the table $T_i[\ ]$ has 2^{t_i} entries, each entry is an h-tuple. (For example, let $h = 16$ and $t_i = 8$. The table $T_i[\ ]$ then has 2^8 entries, 16 bits each, i.e., the total memory storage for this particular table is $2^8 \times 16$ bits = 512 bytes). Finally, the total number of entries for the m tables, denoted by g_b, is

$$g_b = \sum_{i=1}^{m-1} 2^{t_i} + 2^{t_m} = \sum_{i=1}^{m} 2^{t_i} \tag{47}$$

To summarize, for a given polynomial $A(X)$ of degree less than s, let $m, t_1, t_2, ..., t_m$ be such that $s = \sum_{i=1}^{m} t_i$. The term

$$B(X) = \mathrm{R}_{M(X)}\left[A(X)X^h\right]$$

can then be computed using the m tables defined by (46). The total number of entries for these tables is $g_b = \sum_{i=1}^{m} 2^{t_i}$. Further, regardless of whether $s < h$ or $s \ge h$, it can be shown that, using the m tables, the number of operations required for computing $B(X)$ is

$$r_b = 3(m-1) \tag{48}$$

We now consider the special case $t_1 = t_2 = \cdots = t_m = s/m$. The m tables defined in (46) then becomes

$$T_i[A_i] = \mathrm{R}_{M(X)}\left[A_i(X)X^{h+s(m-i)/m}\right]$$

$i = 1, 2, \ldots, m$. Each of the m tables has $2^{s/m}$ entries. From (47), the total number of table entries is

$$g_b = \sum_{i=1}^{m} 2^{s/m} = m2^{s/m} \tag{49}$$

Equations (48) and (49) show tradeoffs between the operation count r_b and the table size g_b. That is, to decrease the table size, we must increase m in $g_b = m2^{s/m}$, and this in turn will increase the operation count $r_b = 3(m-1)$. Thus, smaller (larger) table size g_b will yield larger (smaller) operation count r_b. In particular, when $m = 1$, we have $r_b = 0$ and $g_b = 2^s$. When $m = s$, we have $r_b = 3(s-1)$ and $g_b = 2s$.

Substituting $r_b = 3(m-1)$ into (44), we have

$$e_b = \begin{cases} (24m+24)/s & \text{if } s < h \text{ and } h = 8, 16, 32, 64 \\ (24m+32)/s & \text{if } s < h \text{ and } h \ne 8, 16, 32, 64 \\ 24m/s & \text{if } s = h \\ (24m+8)/s & \text{if } s > h \end{cases} \tag{50}$$

Note that our formulation is a straightforward generalization of [15], which contains the results for the special cases $h = 16$, $s \in \{8, 16\}$, and $m \in \{1, s\}$. Our results [e.g., (49)] also resemble those of [9], which presents in-depth studies of the case $h = 32$.

Note that the function $f(x) = 2^x$ is convex. Given $s = \sum_{i=1}^{m} t_i$, from Jensen's inequality, it can then be shown that $m^{-1} \sum_{i=1}^{m} 2^{t_i} \geq 2^{s/m}$, i.e., $\sum_{i=1}^{m} 2^{t_i} \geq m2^{s/m}$. This implies that, for a given m, the table size g_b in (47) is minimized when $t_1 = t_2 = \cdots = t_m$. Thus, we focus on only this special case ($t_i = s/m$) in this paper.

For example, let $h = 16$, $s = 8$, and $m = 1$. That is, we use a basic 16-bit CRC to protect a message consisting of input bytes ($s = 8$). This CRC is implemented using one lookup table ($m = 1$), which has $g_b = m2^{s/m} = 2^8$ entries. Using (50) with $s < h$, we have $e_b = (24m + 24)/s = 6$. That is, 6 operations are required for computing the check tuple per input byte. These results are recorded in the first row of Fig. 19. Note that, because each table entry has $h = 16$ bits, the total storage is $hg_b = 16 \times 2^8$ bits $= 512$ bytes (which is not shown in the figure). The results for other values of h, s, and m are shown in Fig. 19.

From Fig. 19, we observe the followings. First, the results for the cases $h = 8$ and $h = 16$ are identical for $s = 32, 64$, i.e., they differs only for $s = 8, 16$. This follows directly from (50). Similarly, the results for the cases $h = 16$ and $h = 32$ are identical for $s = 8, 64$, i.e., they differs only for $s = 16, 32$. Although the number of table entries $g_b = m2^{s/m}$ depends on only s and m, the total storage is $hg_b = hm2^{s/m}$, which also depends on h.

Recall from Fig. 9 that the complexity results for bitwise implementation of the basic CRCs vary little over a wide range of s values. In contrast, as seen in Fig. 19, those for table-lookup implementation vary greatly with s. These results can also be used to optimize CRC table-lookup implementation. For example, suppose that $h = 16$. Let us compare the 2 cases: ($s = 8, m = 1$) and ($s = 16, m = 4$) in Fig. 19. In both cases, the required operation count per input byte is $e_b = 6$. However, the first case requires one table of 2^8 entries ($= 16 \times 2^8 = 512$ bytes), while the second case requires 4 tables totaling only 2^6 entries ($= 16 \times 2^6 = 128$ bytes), which is 75% less than the first case. More generally, Fig. 19 shows that, for a given e_b, the total number of table entries g_b is minimized when $s = h$.

		$h=8$	$h=16$	$h=32$	$h=64$	
	m	e_b	e_b	e_b	e_b	g_b
$s = 8$	1	3	6	6	6	2^8
	2	6	9	9	9	2^5
	4	12	15	15	15	2^4
$s = 16$	1	2	1.5	3	3	2^{16}
	2	3.5	3	4.5	4.5	2^9
	4	6.5	6	7.5	7.5	2^6
	8	12.5	12	13.5	13.5	2^5
$s = 32$	1	1	1	0.75	1.5	2^{32}
	2	1.75	1.75	1.5	2.25	2^{17}
	4	3.25	3.25	3	3.75	2^{10}
	8	6.25	6.25	6	6.75	2^7
	16	12.25	12.25	12	12.75	2^6
$s = 64$	1	0.5	0.5	0.5	0.375	2^{64}
	2	0.875	0.875	0.875	0.75	2^{33}
	4	1.625	1.625	1.625	1.5	2^{18}
	8	3.125	3.125	3.125	3	2^{11}
	16	6.125	6.125	6.125	6	2^8
	32	12.125	12.125	12.125	12	2^7

Fig. 19 Complexity results for table-lookup technique for the basic h-bit CRCs (e_b = operation count per input byte, g_b = total number of entries from m tables)

Remark 9. Both g_b and e_b depend on s, h, and m, i.e., we can write $g_b = g_b(s, h, m)$ and $e_b = e_b(s, h, m)$. Consider the 2 special cases: $m = s/2$ and $m = s$. From (49) and (50), it can be shown that $g_b(s, h, s/2) = g_b(s, h, s) = 2s$ and $e_b(s, h, s/2) < e_b(s, h, s)$. That is, these 2 cases yield the same table size, but the case $m = s/2$ always yields lower operation count than the case $m = s$. Thus, the case $m = s$ can be eliminated from our discussion. □

Remark 10. So far, $B(X)$ is computed by either the bitwise technique or the table-lookup technique. However, $B(X)$ can also be computed using both techniques as follows. Recall from (45) that $B(X)$ is the sum of m terms. Suppose that we now use tables to compute the first $m-1$ terms, and use no tables to compute the last term. More precisely, from (45), we have

$$\begin{aligned} B(X) &= \sum_{i=1}^{m-1} \mathrm{R}_{M(X)} \left[A_i(X) X^{(t_{i+1}+\cdots+t_m+h)} \right] + \mathrm{R}_{M(X)} \left[A_m(X) X^h \right] \\ &= \sum_{i=1}^{m-1} T_i[A_i] + \mathrm{R}_{M(X)} \left[A_m(X) X^h \right] \end{aligned}$$

where the $m-1$ tables $T_i[\ ]$ are defined by $T_i[A_i] = \mathrm{R}_{M(X)} \left[A_i(X) X^{(t_{i+1}+\cdots+t_m+h)} \right]$, $1 \le i < m$. Assume now that $\mathrm{R}_{M(X)} \left[A_m(X) X^h \right]$ is computed without using tables. Thus, $B(X)$ can be computed using the 2 techniques at the same time: the table lookup technique (with the $m-1$ tables $T_i[\ ]$) and the bitwise technique (for computing $\mathrm{R}_{M(X)} \left[A_m(X) X^h \right]$ without using tables). In the following, this mixed technique is applied to the fast CRCs to yield small table size when $s \approx h$. □

A.2.2 Fast CRCs

Recall from Section 2.1 that, when implementing an h-bit CRC, we are free to choose the value of s, which is the size of each input tuple $Q_i(X)$. That is, we can choose $s < h$, $s = h$, or $s > h$. Fig. 9 shows that, under bitwise implementation, the fast CRCs are much faster than the basic CRCs for $s \le h$, in the sense that e_f is much smaller than e_b. Further, by comparing Fig. 9 with Fig. 19, we see that the bitwise implementation of the fast CRCs (i.e., $g_f = 0$) is even faster than the table-lookup implementation of the basic CRCs (i.e., $g_b > 0$) in many cases. For example, consider the case $s = h = 32$. Fig. 19 shows that $e_b = 6$ when $g_b = 2^7$ (at $m = 8$), and $e_b = 12$ when $g_b = 2^6$ (at $m = 16$). On the other hand, Fig. 9 shows that $e_f = 3$ when $g_f = 0$. The same figures also show that, although the fast CRC requires no table lookup (i.e., $g_f = 0$) and the basic CRC requires a table of $g_b = 2^{10}$ entries (at $m = 4$), both CRCs have the same operation count $e_f = e_b = 3$.

Recall from (41) and (43) that, under bitwise implementation, e_f is minimized either at $s = h-2$ or at $s = h$. Thus, by choosing s to be at (or near) these optimal values, the fast CRCs require no table lookup (i.e., $g_f = 0$) and still have low operation count (i.e., e_f is small).

We now discuss table-lookup techniques for the fast CRCs generated by $F_h(X) = X^h + X^2 + X + 1$. An obvious technique is to apply the table-lookup technique in Section A.2.1 for the basic CRCs to the fast CRCs by simply letting $M(X) = F_h(X)$. The required total number of table entries is then given by (49), i.e., $g_f = g_b = m2^{s/m}$. In the following, for the case $s \ge h$, we present another table-lookup technique (which is similar to the mixed technique in Remark 10 with $m = 2$) that exploits the special structure of $F_h(X)$ to yield $g_f = 2^{s-h+2}$, which is small when $s \approx h$, e.g., $g_f = 4$ when $s = h$.

Recall that r_f denotes the operation count required for computing $B(X)$. Without using tables (i.e., when $g_f = 0$), r_f is given by (38), i.e., $r_f = 9 + 5.5(s-h)$ for $s \ge h$. We show below that r_f is slightly reduced by using a small lookup table.

Assume that $s \ge h$. According to Fig. 10, we use Algorithm 2 to implement the table-lookup technique for the fast CRCs when $s \ge h$. From Fig. 13, we then have

$$x = \begin{cases} 3 & \text{if } s = h \\ 4 & \text{if } s > h \end{cases}$$

which is inserted in (33) to yield

$$e_f = \begin{cases} 8(3 + r_f)/s & \text{if } s = h \\ 8(4 + r_f)/s & \text{if } s > h \end{cases} \tag{51}$$

where r_f, which denotes the operation count required for computing $B(X)$ via table lookup, is determined in the following.

First, we decompose $A(X)$ into $A_1(X)$ and $A_2(X)$ such that

$$A(X) = A_1(X) X^{h-2} + A_2(X)$$

where degree$(A_1(X)) < s-h+2$ and degree$(A_2(X)) < h-2$. Using (11) with $M(X) = F_h(X)$, we have

$$\begin{aligned} B(X) &= \mathrm{R}_{F_h(X)}\left[A(X)X^h\right] \\ &= \mathrm{R}_{F_h(X)}\left[A(X)(X^h + F_h(X))\right] \\ &= \mathrm{R}_{F_h(X)}\left[A(X)(X^2 + X + 1)\right] \\ &= \mathrm{R}_{F_h(X)}\left[A_1(X)X^{h-2}(X^2+X+1)\right] + A_2(X)(X^2+X+1) \\ &= T_f[A_1] + A_2(X)(X^2+X+1) \end{aligned}$$

where $T_f[\,]$ is the table defined by

$$T_f[A_1] = \mathrm{R}_{F_h(X)}\left[A_1(X)X^{h-2}(X^2+X+1)\right] \tag{52}$$

where A_1 denotes the $(s-h+2)$-tuple that is composed of the binary coefficients of the polynomial $A_1(X)$ of degree less than $s-h+2$. The table $T_f[\,]$ has $g_f = 2^{s-h+2}$ entries, and each entry contains h bits. Using this table, it can be shown that the operation count required for computing $B(X)$ is $r_f = 7$. To summarize, when $s \geq h$, we have

$$r_f = 7, \quad g_f = 2^{s-h+2} \tag{53}$$

Substituting (53) into (51), we then have the following operation count per input byte and the table size for the case $s \geq h$:

$$e_f = \begin{cases} 80/s, & g_f = 4 \qquad \text{if } s = h \\ 88/s, & g_f = 2^{s-h+2} \quad \text{if } s > h \end{cases} \tag{54}$$

i.e., the table size g_f grows exponentially with the difference $s-h$. Thus, this table-lookup technique is not recommended for large $s-h$. To have a small table, we must choose s that is sufficiently close to h. The table size is minimized when $s = h$, which yields $g_f = 4$, i.e., the fast h-bit CRCs now require $e_f = 80/h$ operations per input byte and a small table of only $g_f = 4$ entries. This is 20% lower than the bitwise technique (i.e., $g_f = 0$) that requires $e_f = 96/h$ operations per input byte. Fig. 20 shows the numerical values of (54) for $s, h \in \{8, 16, 32, 64\}$, which vary greatly with h and s. In particular, the table size is large ($g_f \geq 2^{10}$) when $s > h$, but is very small ($g_f = 4$) when $s = h$.

For the special case $s = h$ for which $g_f = 4$, it can be shown that the 4 entries of the table (52) are given by

$$\begin{aligned} T_f[0] &= 0 \\ T_f[1] &= X^{h-1} + X^{h-2} + X^2 + X + 1 \\ T_f[2] &= X^{h-1} + X^3 + 1 \\ T_f[3] &= X^{h-2} + X^3 + X^2 + X \end{aligned}$$

These entries in hexadecimal are shown in Fig. 21.

The table-lookup algorithm for the fast CRC (when $s \geq h$) is given in Fig. 22, where the 2^{s-h+2} entries of the table $T_f[\,]$ defined by (52) are stored in the top part of the algorithm. The C program for the special case $s = h = 16$, which is based on Fig. 22, is given in Fig. 23.

	s	e_f	g_f
$h = 8$	8	10	2^2
	16	5.5	2^{10}
	32	2.75	2^{26}
	64	1.375	2^{58}
$h = 16$	16	5	2^2
	32	2.75	2^{18}
	64	1.375	2^{50}
$h = 32$	32	2.5	2^2
	64	1.375	2^{34}
$h = 64$	64	1.25	2^2

Fig. 20 Complexity results for the fast h-bit CRCs with table lookup, $s \geq h$ (e_f = operation count per input byte, g_f = total number of table entries)

h	$T_f[0]$	$T_f[1]$	$T_f[2]$	$T_f[3]$
8	0	c7	89	4e
16	0	c007	8009	400e
32	0	c0000007	80000009	4000000e
64	0	c000000000000007	8000000000000009	400000000000000e

Fig. 21 Four-entry tables for the fast h-bit CRCs ($s = h$)

```
store $T_f[0], \ldots, T_f[2^{s-h+2} - 1]$;
$B = 0$;
for ($0 \leq i < n$)
    {
    $A = BX^{s-h} + Q_i$;
    $A_1 = (s - h + 2)$ left-hand bits of $A$;
    $A_2 = (h - 2)$ right-hand bits of $A$;
    $B = T_f[A_1] + A_2(X^2 + X + 1)$;
    }
$P = B$;
return $P$;
```

Fig. 22 Table-lookup algorithm for the fast h-bit CRC ($s \geq h$)

```
unsigned short
fast_CRC_table (int n, unsigned short *Q)
{
int            i;
unsigned short A, A1, A2, P;
static unsigned short T[4] =
{0x0, 0xc007, 0x8009, 0x400e};

P = 0;
for (i=0; i<n; i=i+1)             /* 2 */
    {
    A = P ^ Q[i];                 /* 1 */
    A1 = A >> 14;                 /* 1 */
    A2 = A & 0x3fff;              /* 1 */
    P = T[A1]^(A2<<2)^(A2<<1)^A2; /* 5 */
    }

return P;
}
```

Fig. 23 C program for the fast h-bit CRC with table lookup ($s = h = 16$)

APPENDIX B OTHER FAST ERROR-DETECTION CODES

So far, we apply the new technique (15) to $F_h(X) = X^h + X^2 + X + 1$ to yield the fast h-bit CRCs. We now use this same technique to construct some other error-detection codes, which are also fast and have minimum distances $d = 2$, 3, or 4.

Recall from Section 3.3 that the *maximum length* of an error-detection code is defined to be the total bit length at or below which its minimum distance is $d \geq 3$, i.e., beyond which its minimum distance will reduce to $d = 2$. Theorem 2 shows that the maximum length of a CRC is the *period* of its generator polynomial. In the following, n_{b} denotes the total bit length of a code.

B.1 Fast CRCs Generated by Binomials

Consider the h-bit CRC generated by the *binomial* $M(X) = X^h + 1$, which has period h. To avoid triviality, we assume that this CRC includes at least one input bit, i.e., $n_{\mathrm{b}} > h$. From Theorem 2, this CRC then has the minimum distance $d = 2$, i.e., it is a weak code for error detection. This CRC can be implemented via Fig. 3 (for $s < h$) or Fig. 4 (for $s \geq h$). Applying the new technique (15) to $M(X) = X^h + 1$, the term $B(X)$ in these figures is given by

$$B(X) = \begin{cases} A(X) & \text{if } s < h \\ \mathrm{R}_{N(X)}\left[A(X)X^{s-h}\right] & \text{if } s \geq h \end{cases}$$

where $N(X) = (X^h + 1)X^{s-h}$. Note that by choosing $s \leq h$, we have $B(X) = A(X)$, i.e., the polynomial division is eliminated.

Suppose now that $s = h$. The CRC generated by $X^h + 1$ can then be implemented by Fig. 6 with $B(X) = A(X)$. Fig. 6 can be further simplified to yield the following pseudocode for computing the check h-tuple $P(X)$:

```
$P = 0$;
for $(0 \leq i < n)$
    $P = P + Q_i$;
return $P$;
```

which yields

$$P(X) = \sum_{i=0}^{n-1} Q_i(X)$$

i.e., the CRC generated by $M(X) = X^h + 1$ is identical to the bock-parity checksum [4]. From the above pseudocode, it can be shown that computing the check tuple $P(X)$ for this checksum requires $e = 24/s$ operations per input byte. Recall from Section 4 that $e_b = 8(3 + 5.5s)/s$ and $e_f = 96/s$. We then have $e_f/e = 96/24 = 4$ and $e_b/e = 8(3 + 5.5s)/24 = (3 + 5.5s)/3$.

For example, if $s = h = 16$, then computing the check tuple $P(X)$ for the 16-bit bock-parity checksum requires $e = 24/16 = 1.5$ operations per input byte. We then have $e_f/e = 4$ and $e_b/e = (3 + 5.5 \times 16)/3 = 30.33$. Thus, as expected, the bock-parity checksum (which has minimum distance $d = 2$) is substantially faster than the fast and basic CRCs (both of which have minimum distance $d = 4$).

B.2 Fast CRCs Generated by Trimomials

Let C be the CRC generated by the *trinomial* $T_h(X) = X^h + X + 1$. The periods t of the trinomials are given in Fig. 24 for $h \geq 3$. Note that the periods t for the important cases $h =$ 8, 16, 32, 64 are unusually small. Because $T_h(X)$ is a codeword of weight 3, the minimum distance d of this CRC must satisfy $d \leq 3$. From Theorem 2, we then have $d = 3$ if $n_b \leq t$, and $d = 2$ if $n_b > t$. This CRC can be implemented via Fig. 3 (for $s < h$) or Fig. 4 (for $s \geq h$). Applying the new technique (15) to $M(X) = T_h(X)$, the term $B(X)$ in these figures is given by

$$B(X) = \begin{cases} \mathrm{R}_{T_h(X)}\left[A(X)(X+1)\right] & \text{if } s < h \\ \mathrm{R}_{N(X)}\left[A(X)X^{s-h}(X+1)\right] & \text{if } s \geq h \end{cases} \tag{55}$$

where $N(X) = (X^h + X + 1)X^{s-h}$. Remark 1 implies that it is simpler to compute the $B(X)$ in (55) than the $B(X)$ in (16). Thus, the CRC generated by the trinomial $T_h(X) = X^h + X + 1$ is faster than the fast CRC generated by $F_h(X) = X^h + X^2 + X + 1$. However, the former has minimum distance only $d = 3$, whereas the latter has minimum distance $d = 4$. Further, for the important cases of $h = 8, 16, 32, 64$, the maximum length of the faster CRC generated by the trinomial $T_h(X)$ is much shorter than that of the fast CRC generated by $F_h(X)$. For example, the faster 16-bit CRC generated by $T_{16}(X)$ has $d = 3$ and the maximum length of only 255 bits (see Fig. 24), whereas the fast CRC generated by $F_{16}(X)$ has $d = 4$ and the maximum length of $2^{15} - 1 = 32767$ bits (see Section 3.2). Thus, these 2 types of CRCs illustrate tradeoffs between code capability and complexity.

h	period	$\frac{2^h-1}{\text{period}}$
3	7	1
4	15	1
7	127	1
8	63	4.05
15	32767	1
16	255	257
23	2088705	4.02
24	2097151	8
31	2097151	1024
32	1023	4.2×10^6
63	$2^{63} - 1$	1
64	4095	4.5×10^{15}
127	$2^{127} - 1$	1
128	16383	2.1×10^{34}

Fig. 24 The period of trinomial $T_h(X) = X^h + X + 1$

B.3 Fast and Optimal Error-Detection Codes

In the following, we construct codes that are not only fast, but also have optimal error-detecting capability. The h-bit CRC in Section B.2, which is denoted by C and has minimum distance $d = 3$, can be extended to yield a code that has $d = 4$ by adding an overall parity bit to the h-bit CRC. Note that this extended code, denoted by C^*, has $h^* = h + 1$ check bits and is not a CRC. The h-bit CRC has burst-error-detecting capability $b = h$. The following theorem shows that the extended code C^* has burst-error-detecting capability $b = h^* = h + 1$.

Theorem 4. Let C be an h-bit CRC generated by a polynomial $M(X)$ of degree h. Assume that X is not a factor of $M(X)$, i.e., $\gcd(X, M(X)) = 1$, and that $M(X)$ has odd weight, i.e., it has an odd number of terms. Let C^* be the non-CRC code that is obtained by adding an overall parity check bit to C, i.e., C^* has $h^* = h + 1$ check bits. Then C^* detects all error bursts of length $h + 1$ or less, i.e., its burst-error-detecting capability is $b = h + 1$.

Proof. Let $V^*(X)$ be a codeword of C^*. By definition of C^*, we have $V^*(X) = V(X)X + \text{parity}(V(X))$, where $V(X)$ is a codeword of C. Because $V(X)$ is a codeword of the CRC generated by $M(X)$, we have $V(X) = K(X)M(X)$ for some polynomial $K(X)$ (see the proof of Theorem 2). We then have

$$V^*(X) = K(X)M(X)X + \text{parity}(K(X)M(X))$$

Let $E^*(X)$ be an error burst of length $h + 1$ or less, which has the form

$$E^*(X) = X^i(E(X) + 1)$$

where $i \geq 0$, and $E(X)$ is a polynomial such that $E(X) \neq 1$ and $\text{degree}(E(X)) \leq h$. Using proof by contradiction, we now show that $E^*(X)$ cannot be a codeword of C^*. Thus, assume that $E^*(X)$ is a nonzero codeword of C^*, i.e.,

$$X^i(E(X) + 1) = K(X)M(X)X + \text{parity}(K(X)M(X))$$

We consider 2 cases: $i = 0$ and $i > 0$.

Case 1: $i = 0$. We then have

$$E(X) + 1 = K(X)M(X)X + \text{parity}(K(X)M(X))$$

This implies that $\text{parity}(K(X)M(X)) = 1$. Thus, $K(X) \neq 0$, which implies that $\text{degree}(K(X)M(X)X > h$. But we also have $E(X) = K(X)M(X)X$, which implies that $\text{degree}(K(X)M(X)X) \leq h$, which is a contradiction to the previous statement.

Case 2: $i > 0$. We then have $\text{parity}(K(X)M(X)) = 0$. Thus, $X^i(E(X) + 1) = K(X)M(X)X$. Because $\gcd(X, M(X)) = 1$, we must have $X^i = K(X)X$. Thus, $K(X) = X^{i-1}$. We then have $\text{parity}(K(X)M(X)) = \text{parity}(M(X)) = 1$, which is a contradiction to the previous statement that $\text{parity}(K(X)M(X)) = 0$. □

Let t be the period of the polynomial $M(X)$ in Theorem 4. The extended code C^* in Theorem 4 then has $h^* = h + 1$ check bits, the burst-error-detecting capability $b = h^* = h + 1$, the minimum distance

$d = 4$, and the maximum length of $t + 1$ bits. In the following, we show that C^* becomes fast by choosing $M(X) = T_h(X) = X^h + X + 1$, i.e., $M(X)$ is a trinomial.

Thus, let $M(X) = X^h + X + 1$, and $P_{\text{CRC}}(X)$ be the check h-tuple for the CRC generated by this particular $M(X)$. Suppose that $s = h + 1$. Because $s > h$, the check tuple $P_{\text{CRC}}(X)$ can be computed by Algorithm 4 (see Fig. 4), in which the term $B(X)$ is computed by (55), i.e.,

$$\begin{aligned} B(X) &= \mathrm{R}_{N(X)}\left[A(X)X(X+1)\right] \\ &= \mathrm{R}_{N(X)}\left[A(X)(X^2+X)\right] \end{aligned}$$

where $N(X) = (X^h + X + 1)X$, and degree$(A(X)) < s = h + 1$.

Recall from Theorem 4 that the non-CRC code C^* is obtained by adding an overall parity check bit to the above CRC. The overall parity bit of C^* is computed as follows. First, we define

$$W(X) = \sum_{i=0}^{n-1} Q_i(X) + P_{\text{CRC}}(X)X$$

where $Q_0(X), \ldots, Q_{n-1}(X)$ are the input s-tuples. The overall parity bit of C^* is also the parity bit of $W(X)$. The check polynomial of C^* is then

$$P(X) = P_{\text{CRC}}(X)X + \text{parity}(W(X))$$

which is a polynomial of degree $< h + 1$.

Fig. 25 shows an implementation of C^*, which is based on Fig. 4 (with $s = h^* = h + 1$ and $M(X) = X^h + X + 1$) and includes the calculation of the overall parity bit of C^*. Let e^* be the operation count per input byte required for computing the check tuple $P(X)$ for the code C^*. By ignoring the negligible complexity due to the terms outside the loop indexed by i in Fig. 25, it can be shown that $e^* = 96/h^*$. It is shown in (39) of Appendix A that the complexity for the fast h^*-bit CRC is also given by $e_f = 96/h^*$ (when $s = h^*$). Thus, $e^* = e_f$, i.e., the (non-CRC) h^*-bit code C^* is as fast as the fast h^*-bit CRC.

Let $M(X)$ be a primitive polynomial of degree h, i.e., the period of $M(X)$ is $t = 2^h - 1$. Let us now compare the capability and complexity for the following 2 codes, each of which has $h^* = h + 1$ check bits. The first code is the familiar basic CRC generated by $(X + 1)M(X)$, which has $d = 4$, $b = h + 1$, and the maximum length of $2^h - 1$ bits. An example is the well-known CRC-16, which is generated by $M(X) = (X + 1)(X^{15} + X + 1) = X^{16} + X^{15} + X^2 + 1$. Under bitwise implementation, this basic CRC requires $e_b = 45.5$ operations per input byte for computing its check tuple, provided that the input message is composed of 16-tuples, i.e., $s = h = 16$ (see Fig. 9).

The second code is the non-CRC code C^* as described in Theorem 4, which has $d = 4$, $b = h + 1$, and the maximum length of $t + 1 = 2^h$ bits (which is 1 bit longer than the basic $(h + 1)$-bit CRC above). It is well-known that any code that has $h + 1$ check bits and the minimum distance $d = 4$ must satisfy the following 2 constraints: (1) the burst-error detecting capability $b \leq h + 1$ and (2) the maximum length $\leq 2^h$. Thus, the non-CRC code C^* is optimal for error detection in the sense that, with $h^* = h + 1$ check bits and $d = 4$, it has the optimal $b = h^*$ and the optimal maximum length 2^h. In fact, at the maximum length of 2^h bits, the code C^* is a $(2^h, 2^h - h - 1, 4)$ extended Hamming perfect code with the optimal burst-error-detecting capability $b = h + 1$. Also, it is well-known that the undetected error probability of this perfect code is bounded above by $2^{-(h+1)}$.

As shown in Fig. 25, the code C^* is fast when $M(X) = T_h(X) = X^h + X + 1$, i.e., $M(X)$ is a trinomial. It is known that $T_h(X)$ is primitive for some values of h [18], including h = 3, 7, 15, 63, and 127 (i.e., $h + 1$ = 4, 8, 16, 64, and 128). For example, let $h = 15$ and $s = h^* = h + 1 = 16$. Using Fig. 25, it can be shown that the operation count per input bye required for computing the check tuple for the (non-CRC) 16-bit code C^* is $e^* = 96/16 = 6$ (which is much smaller than $e_b = 45.5$ of the basic CRC-16 above). To summarize, the (non-CRC) h^*-bit code C^* (e.g., with h^* = 4, 8, 16, 64, or 128 check bits) constructed from a primitive trinomial and an overall parity check bit has (a) the optimal error-detection capability and (b) a fast bitwise implementation. Note that, as discussed later in Section C.3, other fast and optimal codes can also be constructed from polynomials different from trinomials.

```
$W = 0$;
$B = 0$;
for $(0 \leq i < n)$
    {
    $A = B + Q_i$;
    $B = \mathrm{R}_N \left[A(X^2 + X)\right]$;
    $W = W + Q_i$;
    }
$W = W + B$;
$P = B + \mathrm{parity}(W)$;
return $P$;
```

Fig. 25 Algorithm for computing the fast $(h+1)$-bit non-CRC code from the h-bit CRC (generated by $X^h + X + 1$) and an overall parity bit

APPENDIX C APPLICATION OF THE NEW TECHNIQUE TO GENERAL GENERATOR POLYNOMIALS

So far, we apply the new technique (15) to the polynomials X^h+X^2+X+1, X^h+X+1, and X^h+1 to yield fast CRCs. In this appendix, we apply the same technique to more general generator polynomials, and then determine the conditions under which the new technique is faster than the basic technique. In particular, we show later in Section C.1.2.1 that, when applied to the CRC-64-ISO generated by $X^{64}+X^4+X^3+X+1$, the new technique is 15 times faster than the basic technique. This appendix presents only bitwise algorithms.

Consider an h-bit CRC that is generated by a general polynomial

$$\begin{aligned} F(X) &= X^h + X^{i_k} + X^{i_{k-1}} + \cdots + X^{i_1} + 1 \\ &= X^h + H(X) \end{aligned} \tag{56}$$

where $k > 0$, $h > i_k > i_{k-1} > \cdots > i_1 > 0$, and

$$H(X) = X^{i_k} + X^{i_{k-1}} + \cdots + X^{i_1} + 1 \tag{57}$$

Note that $i_k \geq k > 0$, $i_k = \mathrm{degree}(H(X))$, and $k =$ weight of $(H(X)+1)$. Here, we have $F(X) \neq X^h + 1$ because $i_1 > 0$, i.e., $H(X) \neq 1$. The case $F(X) = X^h + 1$ is already discussed in Section B.1, where it is shown that the CRC reduces to the block-parity checksum.

The h-bit CRC generated by (56) can be computed either by the basic technique (see Definition 1) or by the new technique (15). Recall that CRC complexity refers to the operation count per input byte (denoted by e_b and e_f for the basic and the fast CRCs, respectively) required for computing the CRC check tuple. Again, we assume that the CRCs are implemented in C, and the operations are counted according to rules (R1) and (R2) stated in Appendix A.

C.1 General CRC Generator Polynomials

First, suppose that the basic technique is used to compute the check tuple of the CRC generated by (56), i.e., $B(X)$ is computed as in Definition 1 with $M(X) = F(X)$ in (11). From (37), we have

$$e_b = \begin{cases} 8(4+5.5s)/s & \text{if } s < h \\ 8(3+5.5s)/s & \text{if } s \geq h \end{cases} \tag{58}$$

Next, suppose that the new technique is used to compute the check tuple of the CRC generated by (56). By letting $M(X) = F(X)$ in (15), we have

$$B(X) = \begin{cases} \mathrm{R}_{F(X)}\left[A(X)(X^h + F(X))\right] & \text{if } s < h \\ \mathrm{R}_{N(X)}\left[A(X)(X^s + N(X))\right] & \text{if } s \geq h \end{cases} \tag{59}$$

where $N(X) = F(X)X^{s-h}$ and $\mathrm{degree}(A(X)) < s$. Substituting (56) into (59), we have

$$B(X) = \begin{cases} \mathrm{R}_{F(X)}\left[A(X)H(X)\right] & \text{if } s < h \\ \mathrm{R}_{N(X)}\left[A(X)H(X)X^{s-h}\right] & \text{if } s \geq h \end{cases} \tag{60}$$

To briefly illustrate the main idea, consider the special case $s = h$. Then $B(X) = \mathrm{R}_{F(X)}\left[A(X)X^h\right]$ under the basic technique, and $B(X) = \mathrm{R}_{F(X)}\left[A(X)(X^{i_k} + X^{i_{k-1}} + \cdots + X^{i_1} + 1)\right]$ under the new technique. Intuition suggests that computing $B(X)$ via the new technique is faster than the basic technique if i_k is sufficiently small. More precise conditions on i_k are given in the following.

Let e be the operation count per input byte required for computing the CRC check tuple under the new technique (60). We have $e = e_f$ for the special case $F(X) = F_h(X) = X^h + X^2 + X + 1$. Although Fig. 9 shows that $e_f < e_b$, it may not be the case that $e < e_b$ for the more general polynomial $F(X)$. Thus, in the following, we determine the conditions on $F(X)$ so that $e < e_b$ or $e_b/e > 1$ (i.e., the conditions under which the new technique is faster than the basic technique). Thus, the new technique serves as a faster alternative to the basic technique when $e_b/e > 1$. Before continuing, we present the following remarks, which contain some results that will be used later to determine the operation count required for computing $B(X)$.

Remark 11. Let r' be the number of operations required for computing

$$\begin{aligned} B'(X) &= A(X)(X^{j_n} + \cdots + X^{j_1} + 1) \\ &= A(X)X^{j_n} + \cdots + A(X)X^{j_1} + A(X) \end{aligned}$$

where it is assumed that the tuple $A(X)X^{j_i}$ can be stored in a single computer word. Computing $A(X)X^{j_i}$ is then equivalent to shifting $A(X)$ to the left by j_i bits, which can be done by a single operation on most computers. Thus, for a given $A(X)$, we can compute $B'(X)$ by using n left-shift operations and n addition operations. We then have $r' = 2n$. □

Remark 12. Let r^* be the number of operations required for computing

$$\begin{aligned} B^*(X) &= \mathrm{R}_{M^*(X)}\left[A^*(X)(X^{j_n} + \cdots + X^{j_q} + 1)\right] \\ &= \mathrm{R}_{M^*(X)}\left[A^*(X)X^{j_n}\right] + \cdots + \mathrm{R}_{M^*(X)}\left[A^*(X)X^{j_q}\right] + \mathrm{R}_{M^*(X)}\left[A^*(X)\right] \end{aligned}$$

where $n \geq q$, and $\mathrm{R}_{M^*(X)}\left[A^*(X)X^{j_i}\right] \not\equiv A^*(X)X^{j_i}$, i.e., the polynomial division is needed.

Define

$$\begin{aligned} C_{q-1}(X) &= \mathrm{R}_{M^*(X)}\left[A^*(X)\right] \\ C_q(X) &= \mathrm{R}_{M^*(X)}\left[C_{q-1}(X)X^{j_q}\right] \\ C_{q+1}(X) &= \mathrm{R}_{M^*(X)}\left[C_q(X)X^{j_{q+1}-j_q}\right] \\ &\cdots \\ C_{m+q}(X) &= \mathrm{R}_{M^*(X)}\left[C_{m+q-1}(X)X^{j_{m+q}-j_{m+q-1}}\right] \\ &\cdots \\ C_n(X) &= \mathrm{R}_{M^*(X)}\left[C_{n-1}(X)X^{j_n-j_{n-1}}\right] \end{aligned}$$

Let r_0 be the operation count required for computing $C_{q-1}(X)$. Given $C_{q-1}(X)$, the term $C_q(X) = \mathrm{R}_{M^*(X)}\left[C_{q-1}(X)X^{j_q}\right]$ can be computed with $5.5j_q$ operations, and so on. Given $C_{n-1}(X)$, the final term $C_n(X) = \mathrm{R}_{M^*(X)}\left[C_{n-1}(X)X^{j_n-j_{n-1}}\right]$ can be computed in $5.5(j_n - j_{n-1})$ operations. Thus, computing $C_{q-1}(X), C_q(X), \ldots, C_n(X)$ altogether requires

$$r_0 + 5.5j_q + 5.5(j_{q+1} - j_q) + \cdots + 5.5(j_n - j_{n-1}) = r_0 + 5.5j_n$$

operations. Because $B^*(X) = C_{q-1}(X) + C_q(X) + \cdots + C_n(X)$, the tuple $B^*(X)$ is computed by using $(n - q + 1)$ addition operations. Overall, $B^*(X)$ can be computed with

$$r^* = 5.5j_n + n - q + 1 + r_0$$

operations, where r_0 is the operation count required for computing $\mathrm{R}_{M^*(X)}\left[A^*(X)\right]$. □

C.1.1 Case: $s \geq h$

In this case, according to Fig. 10, the new technique uses Algorithm 4 (shown in Fig. 4), which contains the computation of $B(X)$. From (57) and (60), we have

$$\begin{aligned} B(X) &= \mathrm{R}_{N(X)}\left[A(X)H(X)X^{s-h}\right] \\ &= \mathrm{R}_{N(X)}\left[A^*(X)X^{i_k}\right] + \mathrm{R}_{N(X)}\left[A^*(X)X^{i_{k-1}}\right] + \cdots + \mathrm{R}_{N(X)}\left[A^*(X)X^{i_1}\right] + \mathrm{R}_{N(X)}\left[A^*(X)\right] \end{aligned}$$

where $A^*(X) = A(X)X^{s-h}$. Using Remark 3, it can be shown that $\mathrm{R}_{N(X)}[A^*(X)] = \mathrm{R}_{N(X)}\left[A(X)X^{s-h}\right]$ can be computed with $r_0 = 5.5(s-h)$ operations (see Appendix C). Applying Remark 12 with $M^*(X) = N(X)$, $n = k$, $j_n = i_k$, $q = 1$, and $r_0 = 5.5(s-h)$, the tuple $B(X)$ can be computed with $r = 5.5(s-h+i_k)+k$ operations.

Fig. 13 shows that $x = 3$ for $s \geq h$ under Algorithm 4. By substituting the values of x and r into (33), the operation count per input byte required for computing the check tuple under the new technique is

$$e = \frac{8[3 + 5.5(s-h+i_k) + k]}{s} \tag{61}$$

Using (58) and (61), we have

$$\frac{e_b}{e} = \frac{3 + 5.5s}{3 + 5.5(s-h+i_k) + k} \tag{62}$$

Thus, the new technique is faster than the basic technique when $e_b/e > 1$, i.e., $3+5.5s > 3+5.5(s-h+i_k)+k$, which is equivalent to

$$i_k < h - \frac{k}{5.5}$$

where $i_k = \text{degree}(H(X))$ and $k =$ weight of $(H(X)+1)$.

Remark 13. Suppose that $F(X)$ is either $X^h + X^2 + X + 1$ or $X^h + X + 1$. Then $i_k \leq 2$, i.e., i_k is a very small value. Thus, it is appropriate to use loop unrolling in the calculation of $C_m(X)$ above. Then (61)reduces to $e = 8[3 + 5.5(s-h) + 3.5i_k + k]/s$, and then

$$\frac{e_b}{e} = \frac{3 + 5.5s}{3 + 5.5(s-h) + 3.5i_k + k} \tag{63}$$

Note that it is common to choose $s, h \in \{8, 16, 32, 64\}$, i.e., the typical values of s and h are not very small, even when i_k is very small.

For example, suppose now that $s = h$ and $F(X) = F_h(X) = X^h + X^2 + X + 1$, i.e., $k = i_k = 2$ and $e = e_f$. Substituting $s = h$ and $k = i_k = 2$ into (63), we have

$$\frac{e_b}{e} = \frac{e_b}{e_f} = \frac{3 + 5.5h}{3 + 3.5 \times 2 + 2} = 0.25 + 0.458h$$

as previously shown in (29). □

C.1.2 Case: $s < h$

In this case, according to Fig. 10, the new technique uses Algorithm 3 (shown in Fig. 3), which contains the computation of $B(X)$. From (60), we have $B(X) = \mathrm{R}_{F(X)}[A(X)H(X)]$. From Fig. 13, we have

$$x = \begin{cases} 6 & \text{if } h = 8, 16, 32, 64 \\ 7 & \text{if } h \neq 8, 16, 32, 64 \end{cases}$$

By substituting the values of x into (33), the operation count per input byte required for computing the CRC check tuple under the new technique is

$$e = \begin{cases} 8(6+r)/s & \text{if } h = 8, 16, 32, 64 \\ 8(7+r)/s & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{64}$$

where r is the number of operations required for computing $B(X) = \mathrm{R}_{F(X)}[A(X)H(X)]$. From (58), we have $e_b = 8(4 + 5.5s)/s$ for $s < h$, which is used with (64) to yield

$$\frac{e_b}{e} = \begin{cases} (4+5.5s)/(6+r) & \text{if } h = 8, 16, 32, 64 \\ (4+5.5s)/(7+r) & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{65}$$

where r, which depends on whether $i_k \leq h-s$ or $i_k < h-s$, is computed in the following subsections. As seen below, the condition $i_k \leq h-s$ implies that $B(X) = \mathrm{R}_{F(X)}[A(X)H(X)] = A(X)H(X)$, i.e., the polynomial division is eliminated.

C.1.2.1 Case: $s < h$ *and* $i_k \leq h - s$

Because degree$(A(X)) < s$ and degree$(H(X)) = i_k$, we have degree$(A(X)H(X)) < s + i_k$. The assumption $i_k \leq h - s$ then implies that degree$(A(X)H(X)) < h$. Thus, $B(X) = \mathrm{R}_{F(X)}\left[A(X)H(X)\right] = A(X)H(X)$, i.e., the polynomial division is eliminated. Let r be the number of operations required for computing $B(X) = A(X)H(X)$. Using (57), we have

$$B(X) = A(X)X^{i_k} + A(X)X^{i_{k-1}} + \cdots + A(X)X^{i_1} + A(X)$$

Applying Remark 11, we then have $r = 2k$, which is substituted into (64) and (65) to yield

$$e = \begin{cases} 8(6+2k)/s & \text{if } h = 8, 16, 32, 64 \\ 8(7+2k)/s & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{66}$$

$$\frac{e_b}{e} = \begin{cases} (4+5.5s)/(6+2k) & \text{if } h = 8, 16, 32, 64 \\ (4+5.5s)/(7+2k) & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{67}$$

Thus, the new technique is faster than the basic technique if $e_b/e > 1$, i.e.,

$$4 + 5.5s > \begin{cases} 6+2k & \text{if } h = 8, 16, 32, 64 \\ 7+2k & \text{if } h \neq 8, 16, 32, 64 \end{cases}$$

which is equivalent to

$$k < \begin{cases} 2.75s - 1 & \text{if } h = 8, 16, 32, 64 \\ 2.75s - 1.5 & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{68}$$

where $i_k =$ degree$(H(X))$ and $k =$ weight of $(H(X) + 1)$.

For example, consider the CRC-64-ISO, generated by

$$F(X) = X^{64} + X^4 + X^3 + X + 1$$

Here, we have $h = 64$, $k = 3$, and $i_k = 4$. Assume that $s \leq h - i_k = 60$. Under the new technique, we then have

$$\begin{aligned} B(X) &= \mathrm{R}_{F(X)}\left[A(X)(X^4 + X^3 + X + 1)\right] \\ &= A(X)X^4 + A(X)X^3 + A(X)X + A(X) \end{aligned}$$

i.e., the polynomial division is eliminated. Substituting $k = 3$ into (67), we have

$$\frac{e_b}{e} = \frac{4 + 5.5s}{12}$$

For the special case $s = 32$, we have $e_b/e = 15$, i.e., the new technique is 15 times faster than the basic technique for the CRC-64-ISO. We also have $e_b/e = 15$ when $s = 32$ for a 64-bit CRC generated by a polynomial that has the following more general form

$$F(X) = X^{64} + X^{i_3} + X^{i_2} + X^{i_1} + 1$$

where $32 \geq i_3 > i_2 > i_1 > 0$.

C.1.2.2 Case: $s < h$ *and* $i_k > h - s$

As seen below, the computation of $B(X)$ requires the polynomial division in this case. The assumption $i_k > h - s$ implies that there exists m^* such that $1 \leq m^* \leq k$, $i_{m^*} > h - s$, and $i_j \leq h - s$ for all $j < m^*$. There are 3 subcases to consider.

Case 1: $1 < m^* < k$. By letting $m = m^* - 1$, we have

$$F(X) = X^h + X^{i_k} + \cdots + X^{i_{m+1}} + X^{i_m} + \cdots + X^{i_1} + 1$$

where $h > i_k > i_{m+1} > i_m \geq i_1 > 0$, $i_{m+1} > h - s$, and $i_m \leq h - s$.

Because $i_m \le h - s$, we have degree$(A(X)X^{i_n}) < h$, for $1 \le n \le m$. Thus,

$$\mathrm{R}_{F(X)}\left[A(X)X^{i_n}\right] = A(X)X^{i_n}$$

for $1 \le n \le m$. From (60), we then have

$$B(X) = \mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right] + \cdots + \mathrm{R}_{F(X)}\left[A(X)X^{i_{m+1}}\right] + A(X)X^{i_m} + \cdots + A(X)X^{i_1} + A(X)$$

By letting $A^*(X) = A(X)X^{i_{m+1}}$, we can write

$$B(X) = B_1(X) + B_2(X)$$

where

$$B_1(X) = \mathrm{R}_{F(X)}\left[A^*(X)X^{i_k - i_{m+1}}\right] + \cdots + \mathrm{R}_{F(X)}\left[A^*(X)X^{i_{m+2} - i_{m+1}}\right] + \mathrm{R}_{F(X)}\left[A^*(X)\right]$$

and

$$B_2(X) = A(X)X^{i_m} + \cdots + A(X)X^{i_1} + A(X)$$

Because $\mathrm{R}_{F(X)}\left[A^*(X)\right] = \mathrm{R}_{F(X)}\left[A(X)X^{i_{m+1}}\right] = \mathrm{R}_{F(X)}\left[(A(X)X^{h-s})X^{i_{m+1}-(h-s)}\right]$, the term $\mathrm{R}_{F(X)}\left[A^*(X)\right]$ can be computed with $r_0 = 1 + 5.5[i_{m+1} - (h-s)]$ operations for a given $A(X)$. Using Remark 12, $B_1(X)$ can be computed with

$$\begin{aligned} r_1 &= 5.5(i_k - i_{m+1}) + k - (m+2) + 1 + r_0 \\ &= 5.5[i_k - (h-s)] + k - m \end{aligned}$$

operations. Using Remark 11, $B_2(X)$ can be computed with $r_2 = 2m$ operations. Overall, the number of operations required for computing $B(X)$ is

$$\begin{aligned} r &= r_1 + r_2 + 1 \\ &= 5.5[i_k - (h-s)] + k + m + 1 \end{aligned}$$

which is substituted into (65) to yield

$$\frac{e_b}{e} = \begin{cases} (4 + 5.5s)/(6 + 5.5[i_k - (h-s)] + k + m + 1) & \text{if } h = 8, 16, 32, 64 \\ (4 + 5.5s)/(7 + 5.5[i_k - (h-s)] + k + m + 1) & \text{if } h \ne 8, 16, 32, 64 \end{cases} \tag{69}$$

Thus, the new technique is faster than the basic technique when

$$4 + 5.5s > \begin{cases} 6 + 5.5[i_k - (h-s)] + k + m + 1 & \text{if } h = 8, 16, 32, 64 \\ 7 + 5.5[i_k - (h-s)] + k + m + 1 & \text{if } h \ne 8, 16, 32, 64 \end{cases}$$

which is equivalent to

$$i_k < \begin{cases} h - (3 + k + m)/5.5 & \text{if } h = 8, 16, 32, 64 \\ h - (4 + k + m)/5.5 & \text{if } h \ne 8, 16, 32, 64 \end{cases}$$

where $i_k = $ degree$(H(X))$ and $k = $ weight of $(H(X) + 1)$. Recall that we also assume that $h > i_k > i_{m+1} > i_m \ge i_1 > 0$, $i_{m+1} > h - s$, and $i_m \le h - s$.

For example, consider the CRC-32-IEEE 802.3 generated by

$$F(X) = X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1 \tag{70}$$

i.e., $h = 32$, $k = 13$, $i_k = 26$. Assume that $s = 16$. We then have $m = 10$. Substituting these values into (69) yields $e_b/e = 92/85$, i.e., the new technique is slightly faster than the basic technique.

Case 2: $m^* = 1$. We then have

$$F(X) = X^h + X^{i_k} + \cdots + X^{i_1} + 1$$

where $i_1 > h - s$. We have

$$\begin{aligned} B(X) &= \mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right] + \cdots + \mathrm{R}_{F(X)}\left[A(X)X^{i_2}\right] + \mathrm{R}_{F(X)}\left[A(X)X^{i_1}\right] + A(X) \\ &= \mathrm{R}_{F(X)}\left[A^*(X)X^{i_k-i_1}\right] + \cdots + \mathrm{R}_{F(X)}\left[A^*(X)X^{i_2-i_1}\right] + \mathrm{R}_{F(X)}\left[A^*(X)\right] + A(X) \\ &= B_1(X) + A(X) \end{aligned}$$

where $A^*(X) = A(X)X^{i_1}$ and

$$B_1(X) = \mathrm{R}_{F(X)}\left[A^*(X)X^{i_k-i_1}\right] + \cdots + \mathrm{R}_{F(X)}\left[A^*(X)X^{i_2-i_1}\right] + \mathrm{R}_{F(X)}\left[A^*(X)\right]$$

Because $\mathrm{R}_{F(X)}\left[A^*(X)\right] = \mathrm{R}_{F(X)}\left[A(X)X^{i_1}\right] = \mathrm{R}_{F(X)}\left[(A(X)X^{h-s})X^{i_1-(h-s)}\right]$, the term $\mathrm{R}_{F(X)}\left[A^*(X)\right]$ can be computed with $r_0 = 1 + 5.5[i_1 - (h-s)]$ operations for a given $A(X)$. Using Remark 12, $B_1(X)$ can be computed with

$$\begin{aligned} r_1 &= 5.5(i_k - i_1) + k - 2 + 1 + r_0 \\ &= 5.5[i_k - (h-s)] + k \end{aligned}$$

operations. Thus, the number of operations required for computing $B(X)$ is

$$\begin{aligned} r &= r_1 + 1 \\ &= 5.5[i_k - (h-s)] + k + 1 \end{aligned}$$

which is substituted into (65) to yield

$$\frac{e_b}{e} = \begin{cases} (4+5.5s)/(6+5.5[i_k-(h-s)]+k+1) & \text{if } h = 8, 16, 32, 64 \\ (4+5.5s)/(7+5.5[i_k-(h-s)]+k+1) & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{71}$$

Case 3: $m^* = k$. We then have

$$F(X) = X^h + X^{i_k} + \cdots + X^{i_1} + 1$$

where $i_k > h - s$, and $i_n \leq h - s$ for all $n < k$. We have

$$\begin{aligned} B(X) &= \mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right] + A(X)X^{i_{k-1}} + \cdots + A(X)X^{i_1} + A(X) \\ &= \mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right] + B_2(X) \end{aligned}$$

where

$$B_2(X) = A(X)X^{i_{k-1}} + \cdots + A(X)X^{i_1} + A(X)$$

Because $\mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right] = \mathrm{R}_{F(X)}\left[(A(X)X^{h-s})X^{i_k-(h-s)}\right]$, the term $\mathrm{R}_{F(X)}\left[A(X)X^{i_k}\right]$ can be computed with $r_0 = 1 + 5.5[i_k - (h-s)]$ operations for a given $A(X)$. Using Remark 11, $B_2(X)$ can be computed with $r_2 = 2(k-1)$ operations. Thus, the number of operations required for computing $B(X)$ is

$$\begin{aligned} r &= r_0 + r_2 + 1 \\ &= 5.5[i_k - (h-s)] + 2k \end{aligned}$$

which is substituted into (65) to yield

$$\frac{e_b}{e} = \begin{cases} (4+5.5s)/(6+5.5[i_k-(h-s)]+2k) & \text{if } h = 8, 16, 32, 64 \\ (4+5.5s)/(7+5.5[i_k-(h-s)]+2k) & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{72}$$

For example, consider the CRC-32-IEEE 802.3 generated by (70), i.e., $h = 32$, $k = 13$, $i_k = 26$. Assume that $s = 8$. Substituting these values into (72) yields $e_b/e = 48/43$, i.e., the new technique is slightly faster than the basic technique.

C.2 CRC Generator Polynomials of Weight 4

We now consider the special case $k = 2$, i.e., $F(X)$ is a polynomial of weight 4:

$$F(X) = X^h + X^{i_2} + X^{i_1} + 1$$

where $h > i_2 > i_1 > 0$. In particular, $F(X) = F_h(X) = X^h + X^2 + X + 1$ when $i_2 = 2$ and $i_1 = 1$. Fig. 26 lists some weight-4 polynomials $F(X) = X^h + X^{i_2} + X^{i_1} + 1$, which have periods that are greater than those of $F_h(X)$, for $h \leq 32$. Recall from Theorem 2 that the maximum length of a CRC equals the period of its generator polynomial. In the following, we consider the application of the new technique to weight-4 generator polynomials for CRCs such as CRC-16 and CRC-CCITT. For brevity, we only present the results for $s < h$ (the case $s \geq h$ can be handled similarly). There are 3 cases to consider.

Case 1: $i_2 \leq h - s$ (i.e., $s \leq h - i_2$). Using the new technique, we have $B(X) = \mathrm{R}_{F(X)}\left[A(X)(X^{i_2} + X^{i_1} + 1)\right] = A(X)(X^{i_2} + X^{i_1} + 1)$, i.e., the polynomial division is eliminated. Substituting $k = 2$ into (66), we have

$$e = \begin{cases} 80/s & \text{if } h = 8, 16, 32, 64 \\ 88/s & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{73}$$

By comparing (73) with (39), we have

$$e = e_f \tag{74}$$

for $s \leq h - i_2$. Using $k = 2$ and $s \leq h - i_2$ in (68), it can be shown that the new technique is faster than the basic technique when

$$2 \leq s \leq h - i_2 \tag{75}$$

For example, let $F(X) = X^{32} + X^4 + X + 1$, i.e., $h = 32$ and $i_2 = 4$. It follows from (75) that the new technique is faster the basic technique when $2 \leq s \leq 28$. Under this condition, we have

$$B(X) = A(X)(X^4 + X + 1) \tag{76}$$

i.e., the polynomial division is eliminated. Fig. 26 shows that the 32-bit CRC generated by $F(X)$ has the maximum length of $2{,}147{,}483{,}647 = 2^{31} - 1$ bits ($\approx 268{,}435{,}456$ bytes). Recall from Fig. 5 that the original fast 32-bit CRC, generated by $F_{32}(X) = X^{32} + X^2 + X + 1$, has the maximum length of 2,097,151 $\approx (2^{31} - 1)/1024$ bits ($\approx 262{,}143$ bytes). Thus, the maximum length of the CRC generated by $F(X)$ is substantially larger than that of the fast CRC generated by $F_{32}(X)$. However, (74) shows that these 2 CRCs have identical complexity when $s \leq 28$.

Consider the 12-bit CRC generated by $F(X) = X^{12} + X^3 + X + 1$. Fig. 26 shows that this CRC has the maximum length of 2,046 bits, which is much larger than that of the fast CRC generated by $F_{12}(X) = X^{12} + X^2 + X + 1$, which has the maximum length of only 595 bits (see Fig. 5). However, (74) shows that these 2 CRCs have identical complexity when $s \leq 9$.

Case 2: $i_2 > h - s$ and $i_1 \leq h - s$. Using the new technique, we have $B(X) = \mathrm{R}_{F(X)}\left[A(X)(X^{i_2} + X^{i_1} + 1)\right] = \mathrm{R}_{F(X)}\left[A(X)X^{i_2}\right] + A(X)X^{i_1} + A(X)$. Substituting $k = 2$ into (67) yields

$$\frac{e_b}{e} = \begin{cases} (4 + 5.5s)/(10 + 5.5[i_2 - (h - s)]) & \text{if } h = 8, 16, 32, 64 \\ (4 + 5.5s)/(11 + 5.5[i_2 - (h - s)]) & \text{if } h \neq 8, 16, 32, 64 \end{cases}$$

For example, consider the CRC-CCITT generated by $F(X) = X^{16} + X^{12} + X^5 + 1$, i.e., $h = 16$, $i_2 = 12$, and $i_1 = 5$. Assume that $s = 8$. We then have $e_b/e = (4 + 5.5 \times 8)/(10 + 5.5 \times 4) = 48/32 = 1.5$. Thus, for the 16-bit CRC-CCITT, the new technique is 50% faster than the basic technique.

Next, consider the CRC-16 generated by $F(X) = X^{16} + X^{15} + X^2 + 1$, i.e., $h = 16$, $i_2 = 15$, and $i_1 = 2$. Assume also that $s = 8$. We then have $e_b/e = (4 + 5.5 \times 8)/(10 + 5.5 \times 7) = 48/48.5$. Thus, for the CRC-16, the new technique is slightly slower than the basic technique.

Case 3: $i_1 > h - s$. Using the new technique, we have $B(X) = \mathrm{R}_{F(X)}\left[A(X)(X^{i_2} + X^{i_1} + 1)\right] = \mathrm{R}_{F(X)}\left[A(X)X^{i_2}\right] + \mathrm{R}_{F(X)}\left[A(X)X^{i_1}\right] + A(X)$. Substituting $k = 2$ into (71) yields

$$\frac{e_b}{e} = \begin{cases} (4 + 5.5s)/(9 + 5.5[i_2 - (h - s)]) & \text{if } h = 8, 16, 32, 64 \\ (4 + 5.5s)/(10 + 5.5[i_2 - (h - s)]) & \text{if } h \neq 8, 16, 32, 64 \end{cases}$$

$X^h + X^{i_2} + X^{i_1} + 1$	period	$\frac{2^{h-1}-1}{\text{period}}$
$X^5 + X^3 + X + 1$	15	1
$X^7 + X^3 + X^2 + 1$	62	1.01613
$X^7 + X^4 + X^2 + 1$	63	1
$X^9 + X^5 + X^3 + 1$	255	1
$X^{10} + X^3 + X^2 + 1$	511	1
$X^{11} + X^3 + X + 1$	1023	1
$X^{12} + X^3 + X + 1$	2046	1.00049
$X^{12} + X^7 + X^2 + 1$	2047	1
$X^{15} + X^3 + X^2 + 1$	16382	1.00006
$X^{15} + X^5 + X^3 + 1$	16383	1
$X^{17} + X^3 + X + 1$	63457	1.03275
$X^{17} + X^4 + X^3 + 1$	65534	1.00002
$X^{17} + X^{10} + X^4 + 1$	65535	1
$X^{18} + X^5 + X^2 + 1$	98301	1.33336
$X^{18} + X^5 + X^4 + 1$	131071	1
$X^{19} + X^3 + X^2 + 1$	262142	1
$X^{19} + X^5 + X^3 + 1$	262143	1
$X^{20} + X^4 + X^3 + 1$	521985	1.00441
$X^{20} + X^7 + X^5 + 1$	524286	1
$X^{20} + X^{11} + X^2 + 1$	524287	1
$X^{21} + X^3 + X + 1$	1048575	1
$X^{22} + X^3 + X + 1$	491460	4.26719
$X^{22} + X^3 + X^2 + 1$	2094081	1.00147
$X^{22} + X^7 + X^4 + 1$	2097151	1
$X^{23} + X^3 + X + 1$	4161409	1.0079
$X^{23} + X^6 + X + 1$	4194300	1
$X^{23} + X^7 + X^6 + 1$	4194302	1
$X^{23} + X^8 + X^2 + 1$	4194303	1
$X^{25} + X^3 + X + 1$	4194303	4
$X^{25} + X^4 + X + 1$	7864260	2.13335
$X^{25} + X^4 + X^3 + 1$	12070842	1.3899
$X^{25} + X^5 + X + 1$	16766977	1.00061
$X^{25} + X^6 + X^3 + 1$	16777212	1
$X^{25} + X^9 + X^2 + 1$	16777214	1
$X^{25} + X^{14} + X^2 + 1$	16777215	1
$X^{26} + X^3 + X + 1$	32505732	1.03226
$X^{26} + X^4 + X + 1$	33554431	1
$X^{27} + X^3 + X^2 + 1$	67108862	1
$X^{27} + X^5 + X + 1$	67108863	1
$X^{28} + X^3 + X + 1$	97517382	1.37635
$X^{28} + X^5 + X^2 + 1$	134217727	1
$X^{29} + X^{11} + X + 1$	268435455	1
$X^{30} + X^3 + X + 1$	536870908	1
$X^{30} + X^7 + X^6 + 1$	536870911	1
$X^{31} + X^3 + X^2 + 1$	50133510	21.4176
$X^{31} + X^4 + X + 1$	1073213442	1.00049
$X^{31} + X^6 + X^2 + 1$	1073602561	1.00013
$X^{31} + X^6 + X^3 + 1$	1073741822	1
$X^{31} + X^{12} + X^2 + 1$	1073741823	1
$X^{32} + X^3 + X + 1$	21691754	99
$X^{32} + X^3 + X^2 + 1$	22362795	96.0293
$X^{32} + X^4 + X + 1$	2147483647	1

Fig. 26 The period of $X^h + X^{i_2} + X^{i_1} + 1$

C.3 CRC Generator Polynomials of Weight 3

We now consider the special case $k = 1$, i.e., $F(X)$ is a polynomial of weight 3: $F(X) = X^h + X^{i_1} + 1$. By defining $i = i_1$, we have

$$F(X) = X^h + X^i + 1 \tag{77}$$

where $h > i > 0$. Note that $F(X) = T_h(X) = X^h + X + 1$ for the special case $i = 1$. Fig. 27 lists some weight-3 polynomials along with their periods, for $h \leq 31$. In Section B.2, the fast h^*-bit perfect codes are constructed from the CRCs generated by $T_h(X)$ for $h^* = 4, 8, 16, 64, 128$, where $h^* = h+1$. In the following, we show that other fast perfect codes can also be constructed from CRCs generated by weight-3 polynomials.

Let C be the h-bit CRC generated by $F(X)$ in (77). Recall from Theorem 2 that the maximum length of C equals the period of $F(X)$. Assume that $s \leq h - i$. Using the new technique, we have $B(X) = \mathrm{R}_{F(X)}\left[A(X)(X^i + 1)\right] = A(X)(X^i + 1)$, i.e., the polynomial division is eliminated. Let e be the operation count per input byte required for computing the check tuple $P(X)$ of the h-bit CRC C. Then e is given by (66).

Let C^* be the non-CRC code that is constructed by adding an overall parity bit to the h-bit CRC C, and $P^*(X)$ be the check tuple of C^*. Let e^* be the operation count per input byte required for computing $P^*(X)$. Note that $P^*(X)$ has $h+1$ bits, which can be computed by an algorithm that is similar to Fig. 25. Using (66) and Fig. 25, it can then be shown that

$$e^* = \begin{cases} 8(7+2k)/s & \text{if } h = 8, 16, 32, 64 \\ 8(8+2k)/s & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{78}$$

Substituting $k = 1$ into (78), we have

$$e^* = \begin{cases} 72/s & \text{if } h = 8, 16, 32, 64 \\ 80/s & \text{if } h \neq 8, 16, 32, 64 \end{cases} \tag{79}$$

Let us now compare the speed of the $(h+1)$-bit code C^* with that of the fast $(h+1)$-bit CRC generated by $F_{h+1}(X) = X^{h+1} + X^2 + X + 1$. From (39), we have

$$e_f = \begin{cases} 80/s & \text{if } h+1 = 8, 16, 32, 64 \\ 88/s & \text{if } h+1 \neq 8, 16, 32, 64 \end{cases} \tag{80}$$

for $s < h$. By comparing (79) with (80), we have

$$e^* \leq e_f \tag{81}$$

for $s \leq h - i$. Thus, (81) shows that the $(h+1)$-bit non-CRC code C^* is at least as fast as the fast $(h+1)$-bit CRC generated by $F_{h+1}(X)$, i.e., C^* is also a fast code for $s \leq h - i$. Further, at its maximum length, the code C^* is the $(2^h, 2^h - h - 1, 4)$ extended Hamming perfect code, provided that $F(X) = X^h + X^i + 1$ is a primitive polynomial (i.e., its period is $2^h - 1$).

For example, let $F(X) = X^{11} + X^2 + 1$, i.e., $h = 11$ and $i = 2$. Fig. 27 shows that $F(X)$ is primitive. Let C be the 11-bit CRC generated by $F(X)$. The non-CRC code C^*, which is constructed by adding an overall parity bit to C, is the $(2048, 2036, 4)$ extended Hamming perfect code. Note that both C and C^* are fast if we choose $s \leq h - i = 9$. Suppose that we choose $s = 8$. From (79) and (80), we then have $e^* = 80/8 = 10$ and $e_f = 88/8 = 11$, i.e., $e^* < e_f$. Thus, for $s = 8$, the non-CRC 12-bit code C^* is faster than the fast 12-bit CRC generated by $F_{12}(X) = X^{12} + X^2 + X + 1$. Further, the maximum length of the non-CRC code (which is 2,048 bits) is also much longer than that of the fast CRC generated by $F_{12}(X)$ (which is 595 bits), and 2 bits longer than that of the CRC generated by $F(X) = X^{12} + X^3 + X + 1$ (which is 2,046 bits, as discussed in Section C.2).

Similarly, we can construct a 32-bit extended Hamming perfect code C^* by adding an overall parity bit to the CRC C generated by $F(X) = X^{31} + X^3 + 1$ (see Fig. 27). We have $h = 31$ and $i = 3$. Both C and C^* are fast if we choose $s \leq h - i = 28$.

$X^h + X^i + 1$	period	$\frac{2^h-1}{\text{period}}$
$X^3 + X + 1$	7	1
$X^4 + X + 1$	15	1
$X^5 + X + 1$	21	1.47619
$X^5 + X^2 + 1$	31	1
$X^6 + X + 1$	63	1
$X^7 + X + 1$	127	1
$X^8 + X + 1$	63	4.04762
$X^8 + X^3 + 1$	217	1.17512
$X^9 + X + 1$	73	7
$X^9 + X^2 + 1$	465	1.09892
$X^9 + X^4 + 1$	511	1
$X^{10} + X + 1$	889	1.15073
$X^{10} + X^3 + 1$	1023	1
$X^{11} + X + 1$	1533	1.33529
$X^{11} + X^2 + 1$	2047	1
$X^{12} + X + 1$	3255	1.25806
$X^{13} + X + 1$	7905	1.03618
$X^{13} + X^3 + 1$	8001	1.02375
$X^{14} + X + 1$	11811	1.3871
$X^{15} + X + 1$	32767	1
$X^{16} + X + 1$	255	257
$X^{16} + X^3 + 1$	57337	1.14298
$X^{16} + X^7 + 1$	63457	1.03275
$X^{17} + X + 1$	273	480.114
$X^{17} + X^2 + 1$	114681	1.14292
$X^{17} + X^3 + 1$	131071	1
$X^{18} + X + 1$	253921	1.03238
$X^{18} + X^7 + 1$	262143	1
$X^{19} + X + 1$	413385	1.26828
$X^{19} + X^3 + 1$	491505	1.0667
$X^{19} + X^6 + 1$	520065	1.00812
$X^{20} + X + 1$	761763	1.37651
$X^{20} + X^3 + 1$	1048575	1
$X^{21} + X + 1$	5461	384.023
$X^{21} + X^2 + 1$	2097151	1
$X^{22} + X + 1$	4194303	1
$X^{23} + X + 1$	2088705	4.01618
$X^{23} + X^2 + 1$	7864305	1.06667
$X^{23} + X^5 + 1$	8388607	1
$X^{24} + X + 1$	2097151	8
$X^{24} + X^5 + 1$	16766977	1.00061
$X^{25} + X + 1$	10961685	3.06107
$X^{25} + X^2 + 1$	25165821	1.33333
$X^{25} + X^3 + 1$	33554431	1
$X^{26} + X + 1$	298935	224.493
$X^{26} + X^3 + 1$	2094081	32.0469
$X^{26} + X^5 + 1$	67074049	1.00052
$X^{27} + X + 1$	125829105	1.06667
$X^{27} + X^8 + 1$	133693185	1.00392
$X^{28} + X + 1$	17895697	15
$X^{28} + X^3 + 1$	268435455	1
$X^{29} + X + 1$	402653181	1.33333
$X^{29} + X^2 + 1$	536870911	1
$X^{30} + X + 1$	10845877	99
$X^{30} + X^7 + 1$	1073215489	1.00049
$X^{31} + X + 1$	2097151	1024
$X^{31} + X^2 + 1$	22362795	96.0293
$X^{31} + X^3 + 1$	2147483647	1

Fig. 27 The period of $X^h + X^i + 1$

APPENDIX D CRC PARALLEL IMPLEMENTATION

Given a CRC, which is generated by a polynomial $M(X)$ of degree h, our goal is to compute the check h-tuple $P(X)$ to protect an input message $U(X) = (Q_0(X), \ldots, Q_{n-1}(X))$, where $Q_i(X)$ is an s-tuple.

So far, it is implicitly assumed that the CRC algorithms are for sequential implementation. That is, the entire input message $U(X)$ is supplied to a single processor of a computer, and the output $P(X)$ is then computed by this same processor. Following the technique in [6], we can modify these CRC algorithms for parallel implementation on k different processors of a computer, $k > 1$, as follows.

First, the input message $U(X)$ is divided into k sub-messages $E_0(X), \ldots, E_{k-1}(X)$, i.e.,

$$U(X) = (E_0(X), \ldots, E_{k-1}(X))$$

where $E_i(X)$ consists of n_i s-tuples. Thus, $n = n_0 + \cdots + n_{k-1}$. Define

$$W_i(X) = \mathrm{R}_{M(X)}\left[X^{(n_{i+1}+\cdots+n_{k-1})s}\right] \tag{82}$$

for $0 \le i \le k-2$, and $W_{k-1}(X) = 1$. Note that $W_i(X)$ is computed from $X^{(n_{i+1}+\cdots+n_{k-1})s}$, which is used to determine the relative position of sub-message $E_i(X)$ in $U(X)$ (see Remark 14).

Next, for each $i = 0, 1, \ldots, k-1$, input sub-message $E_i(X)$ is supplied to processor i, which is used to compute the following h-tuples:

$$P_i(X) = \mathrm{R}_{M(X)}\left[E_i(X)X^h\right] \tag{83}$$

$$Z_i(X) = \mathrm{R}_{M(X)}\left[P_i(X)W_i(X)\right] \tag{84}$$

where $W_i(X)$ is defined by (82). Note that $P_i(X)$ is the CRC check tuple computed by processor i for sub-message $E_i(X)$. For each $i = 0, 1, \ldots, k-1$, we assume that processor i computes $P_i(X)$ and $Z_i(X)$ in (83) and (84), independent of other processors, i.e., the computation is done in parallel by the k processors.

Theorem 5. The tuples $Z_i(X), 0 \le i < k$, which are computed in parallel by the k processors, are combined to yield the final CRC check h-tuple $P(X)$ for the entire input message $U(X)$, i.e.,

$$P(X) = \sum_{i=0}^{k-1} Z_i(X) \tag{85}$$

Proof. In polynomial notation, we have

$$U(X) = \sum_{i=0}^{k-2} E_i(X)X^{(n_{i+1}+\cdots+n_{k-1})s} + E_{k-1}(X)$$

The CRC check tuple $P(X)$ for $U(X)$ then becomes

$$\begin{aligned} P(X) &= \mathrm{R}_{M(X)}\left[U(X)X^h\right] \\ &= \sum_{i=0}^{k-2} \mathrm{R}_{M(X)}\left[E_i(X)X^hX^{(n_{i+1}+\cdots+n_{k-1})s}\right] + \mathrm{R}_{M(X)}\left[E_{k-1}(X)X^h\right] \\ &= \sum_{i=0}^{k-2} \mathrm{R}_{M(X)}\left[P_i(X)W_i(X)\right] + P_{k-1}(X) \\ &= \sum_{i=0}^{k-1} \mathrm{R}_{M(X)}\left[P_i(X)W_i(X)\right] \\ &= \sum_{i=0}^{k-1} Z_i(X) \end{aligned}$$

□

We now determine the total CRC computation time, denoted by t_{total}, for the parallel technique. First, let t_{W_i}, t_{P_i}, and t_{Z_i} be the times for processor i to compute $W_i(X)$, $P_i(X)$, and $Z_i(X)$, respectively. Let t_P be the time for the computer to compute the summation (85). We can consider t_{W_i}, t_{Z_i}, and t_P as the overhead costs for the CRC parallel implementation. Because the k processors compute (83) and (84) in parallel, the total time for the computer to compute the final CRC check tuple $P(X)$ is

$$t_{\text{total}} = t_{W_i} + \max\{t_{P_i} + t_{Z_i}, 0 \leq i < k\} + t_P \tag{86}$$

We now determine the speedup factor for the parallel technique under the following ideal conditions: (a) the k processors have identical computational capability, (b) the sub-messages $E_i(X)$ have the same length, i.e., $n_i = n/k$, and (c) the overhead costs t_{W_i}, t_{Z_i}, and t_P are negligible compared to t_{P_i}, i.e., $t_{W_i} + t_{P_i} + t_{Z_i} + t_P \approx t_{P_i}$ (see Remark 14). From (86), we then have $t_{\text{total}} \approx t_{P_i} \approx t_U/k$, where t_U denotes the time for a single processor to compute the CRC check tuple $P(X)$ for the entire message $U(X)$, i.e., t_U is the CRC computational time for sequential implementation. Thus, under the ideal conditions, the speedup factor is approximately k for parallel implementation.

Remark 14. Under the CRC parallel implementation, processor i computes $W_i(X)$, $P_i(X)$, and $Z_i(X)$ as given in (82)–(84), $i = 1, \ldots, k-1$. These tuples can be computed as follows. First, it can be shown from (82) that

$$W_i(X) = \mathrm{R}_{M(X)}\left[X^{n_{i+1}s} W_{i+1}(X)\right]$$

with $W_{k-1} = 1$. Thus, once $W_{i+1}(X)$ is known, $W_i(X)$ can be computed in $\mathrm{O}(n_{i+1}s)$ steps (by Remark 1). We can also write $W_i(X) = \mathrm{R}_{M(X)}\left[X^{n_{i+1}s-h} W_{i+1}(X) X^h\right]$, i.e., we can view $W_i(X)$ as the output check tuple of the CRC generated by $M(X)$ when $X^{n_{i+1}s-h} W_{i+1}(X)$ is the input tuple. Thus, $W_i(X)$ can be computed by either the CRC basic technique or the CRC new technique. Suppose now that $n_0, \ldots, n_{k-1}$ are known and fixed. The tuples $W_0(X), W_1(X), \ldots, W_{k-1}$ can then be stored in a table defined by $T[i] = W_i(X)$, $i = 0, 1, \ldots, k-1$ (cf. [6]). Next, processor i can use either the basic technique or the new technique to compute the (partial) CRC check tuple $P_i(X)$. Further, using the technique "Mimic long multiplication as done by hand" in [11, p. 90], it can be shown that the tuple $Z_i(X) = \mathrm{R}_{M(X)}\left[P_i(X) W_i(X)\right]$ can be computed in $\mathrm{O}(h)$ steps. Finally, once $Z_0(X), \ldots, Z_{k-1}(X)$ are computed by the k processors, their summation in (85) can be quickly computed. Thus, for a sufficiently long sub-message $E_i(X)$ along with the use of table lookup for determining $W_i(X)$, the computational complexity of $P_i(X)$ is much greater than that of $W_i(X)$, $Z_i(X)$, and the summation (85), i.e., $t_{P_i} >> t_{W_i}, t_{Z_i}, t_P$. □